Android Dalvik虚拟机结构及机制剖析

——第1卷 Dalvik虚拟机结构剖析

张国印 吴艳霞 编著



清华大学出版社

Android Dalvik 虚拟机结构及机制剖析

——第 1 卷　Dalvik 虚拟机结构剖析

张国印　吴艳霞　编著

清 华 大 学 出 版 社
北　京

内 容 简 介

本系列丛书共分2卷，本书为第1卷，是一本以情景方式对Android的源代码进行深入分析的书，内容广泛，主要从Dalvik虚拟机整体结构、获取和编译Dalvik虚拟机的源码、源码分析辅助工具使用、.dex文件及Dalvik字节码格式解析、Dalvik虚拟机下的系统工具介绍及Dalvik虚拟机执行流程简述等方面进行阐述，帮助读者从宏观上了解Dalvik虚拟机的架构设计，为有兴趣阅读Dalvik虚拟机源码的读者提供必要的入门指导。

第1卷共6章：第1章为准备工作，在这一章中主要介绍了Dalvik虚拟机的功用、分析Dalvik源码所用到的主要方法以及如何搭建Dalvik源码分析环境；第2章为源码分析辅助工具介绍，包括Vim、Doxygen、GDBSERVER等；第3章为Dex文件以及Dalvik字节码格式分析；第4章为系统工具介绍，在这一章中主要介绍了Dalvik虚拟机的一些重要系统工具，通过对系统工具的介绍，让读者对虚拟机内部的实现机制更加清晰；第5章为Dalvik虚拟机执行流程简述，通过这一章的介绍，旨在让读者对Dalvik虚拟机的整体功能架构有一个宏观的认识，为后续进一步掌握各个功能模块的原理功能做好相应的知识铺垫；第6章为调试支撑模块，在这一章中主要介绍了调试支撑模块的基本原理。

通过阅读本书，让读者了解Dalvik虚拟机在Android应用程序运行过程中所扮演的重要角色及其不可替代的价值；同时对Android应用程序的执行过程有更加细致的了解，可以帮助读者优化自己编写的应用程序，更加合理地设计应用程序结构，有效提高应用程序的运行速度。

图书在版编目(CIP)数据

Android Dalvik虚拟机结构及机制剖析. 第1卷　Dalvik虚拟机结构剖析/张国印，吴艳霞编著. --北京：清华大学出版社，2014
ISBN 978-7-302-36103-9

Ⅰ. ①A…　Ⅱ. ①张… ②吴…　Ⅲ. ①移动终端－应用程序－程序设计－虚拟处理机－研究
Ⅳ. ①TP338

中国版本图书馆CIP数据核字(2014)第069719号

责任编辑：袁勤勇　薛　阳
封面设计：
责任校对：李建庄
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦A座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　张**：7.5　　**字　　数**：187千字
版　　次：2014年11月第1版　　**印　　次**：2014年11月第1次印刷
印　　数：1～　000
定　　价：　.00元

产品编号：057014-01

前言

随着移动互联网的不断发展,业务移动化已逐渐被人们接受,移动电子商务、移动办公、移动生活越发深入人心。作为目前市场占有率最高的 Android 操作系统,当之无愧受到广大程序开发人员的青睐。Android 是由 Google 公司基于移动设备而开发的嵌入式系统,具有优良的性能表现以及较低的硬件配置需求,因此使其迅速成为目前移动终端之上的主流操作系统。这种优势的体现主要得益于 Google 对作为 Android 系统基石的 Dalvik 虚拟机所做出的大量优化。对于高阶程序开发人员来说,要想让自己开发的应用程序在数十万应用程序中脱颖而出,就必须掌握整个 Android 系统运行时环境,这其中最为关键的就是 Dalvik 虚拟机。

本书详细地介绍了 Dalvik 虚拟机的结构及其运行机制,尤其针对类数据加载、内存管理、本地方法、反射机制、解释器、即时编译等关键功能模块的设计原理、功能架构以及执行流程进行了介绍,并结合关键代码加以细致讲解。力求让读者了解 Dalvik 虚拟机是如何在底层对 Android 应用程序进行解释执行,并可以结合 Dalvik 虚拟机技术特性对自己的应用程序加以优化改进,以达到进一步提高应用程序安全性、稳定性、高效性的目的。

全书共分为 6 章:

第 1 章为准备工作,主要介绍 Dalvik 虚拟机的定义以及它的功用,分析 Dalvik 源码所用到的主要方法以及如何搭建 Dalvik 源码分析环境。

第 2 章为源码分析辅助工具介绍,主要介绍一些辅助源码分析的工具,包括 Vim、Doxygen、GDBSERVER,并介绍了其使用的方法,为后期的阅读和分析打下基础。

第 3 章为 Dex 文件以及 Dalvik 字节码格式分析,主要介绍 Dex 文件中所涉及的各个数据结构以及相关函数的具体定义,并结合一个 Dex 文件实例对原理内容进行讲解。同时还对 Dalvik 字节码进行了全面的介绍,主要包括字节码设计、字节码格式等内容。另外,在这章的最后还对 Dex 文件的优化产物 Odex 文件功能原理与实际应用进行了简单的介绍,为后续进一步深入讨论 Dex 文件的优化机制做好相关准备。

第 4 章为系统工具介绍,主要介绍 Dalvik 虚拟机的一些重要系统工具,这些工具主要应用于 Dex 文件优化,封装的 apk 文件进行或对 Dex 进行反编译,调试分析 Android 程序源码内存泄漏问题,分析 Android 程序运行过程中生成的 trace 文件等。通过对系统工具的介绍,让读者更清楚虚拟机内部的实现机制。

第 5 章为 Dalvik 虚拟机执行流程简述,主要介绍 Dalvik 虚拟机的整体执行流程以及各个模块所扮演的功能角色。通过这一章的介绍,旨在让读者对 Dalvik 虚拟机的整体功能架构有一个宏观的认识,为后续进一步掌握各个功能模块的原理功能做好相应的知识铺垫。

第 6 章为调试支撑模块,主要介绍调试支撑模块的基本原理,随后,着重介绍 DDM 协

议、JDWP 协议、Debugger 调试器三者的原理及实现，以帮助读者更加清晰地理解调试支撑这部分内容。

本书主要由哈尔滨工程大学张国印、吴艳霞编写，参与本书编写和校核工作的还有汪永峰、王彦璋、谢东良、于成、张婷婷、许圣明、苗施亮、檀凯，这里对他们的辛苦工作表示衷心的感谢。

本书主要是针对高级 Android 应用开发工程师、Android 系统开发工程师、Android 移植工程师及对 Android Dalvik 虚拟机源码实现感兴趣的读者。

作　者

2014 年 6 月

目 录

第 1 章 准备工作

本章主要内容

✍你知道 Dalvik 虚拟机吗?

✍开发者有必要了解 Dalvik 虚拟机吗?

✍如何分析 Dalvik 虚拟机源码?

✍如何搭建源码分析环境?

随着移动互联网的不断发展,业务移动化已逐渐被人们接受,移动电子商务、移动办公、移动生活越发深入人心。智能手机间的竞争已从硬件的竞争中逐渐走出,取而代之的是操作系统的竞争。作为目前市场占有率最高的 Android 操作系统,受到广大程序开发人员的青睐。对于程序开发人员来说,如果能使自己开发的应用程序在数十万应用程序中脱颖而出,无疑是对自身能力最好的肯定。要想达到这种水平,必须深入理解应用开发的各个细节,不仅包括用户体验,还包括代码的质量和性能。想要提高 Android 应用程序的执行效率,就一定要深入理解 Android 应用程序是如何执行的,并且对于整个 Android 系统运行时环境的学习也是十分必要的,这就不得不提到 Dalvik 虚拟机。

1.1 本章概述

当你翻开这本书时,想必一定多次见到过如图 1.1 所示的 Android 系统架构图,在 Android 运行时环境部分(Android Runtime)可以找到接下来将要介绍的 Dalvik 虚拟机(Dalvik Virtual Machine)。

1.1.1 什么是 Dalvik 虚拟机

Dalvik 虚拟机是 Google 等厂商合作开发的 Android 移动设备平台的核心组成部分之一。Dalvik 由 Dan Bornstein 编写,名字来源于他的祖先曾经居住过的名叫 Dalvik 的小渔村,村子位于冰岛。

很多人认为 Dalvik 虚拟机就是一个 Java 虚拟机,因为 Android 的编程语言恰恰就是 Java 语言。但是这种说法并不准确,因为 Dalvik 虚拟机并不是按照 Java 虚拟机的规范来实现的,两者并不兼容,对于这一点后面还会介绍。

那么到底什么是 Dalvik 虚拟机呢? 首先,它是一个虚拟机,也就是一个虚构出来的计算机,是通过在实际的计算机上仿真模拟各种计算机功能来实现的。它有自己完善的硬件架构,如处理器、堆栈、寄存器等,还具有相应的指令系统。第二,这台虚拟出来的计算机主要负责运行 Android 应用程序,它是 Android 应用程序中 Java 代码的运行基础。其指令集

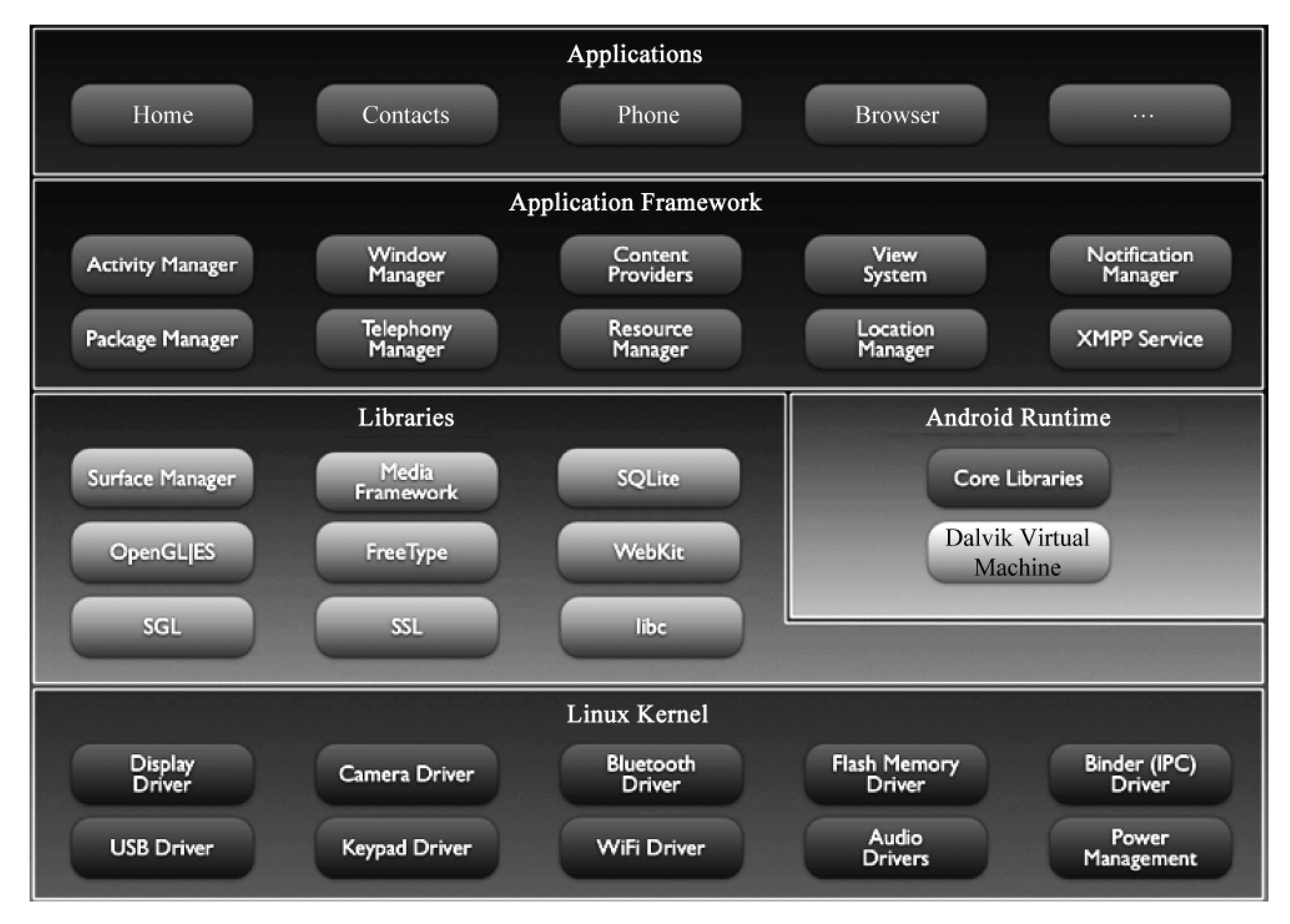


图 1.1 Android 系统架构图

基于寄存器架构，执行其特有的文件格式——Dex 字节码来完成对象生命周期管理、堆栈管理、线程管理、安全异常管理、垃圾回收等重要功能。它的核心内容是实现库(libdvm. so)，大体由 C 语言实现。依赖于 Linux 内核的一部分功能——线程机制、内存管理机制、能高效使用内存以及在低速 CPU 上表现出的高性能。每一个 Android 应用在底层都会对应一个独立的 Dalvik 虚拟机实例，其代码在虚拟机的解释下得以执行。

Android 系统架构可以分为 4 层，分别是应用程序层、应用程序框架层(Framework)，核心库层与 Dalvik VM 以及 Linux 内核。Dalvik VM 和核心库在同一层级，作为承上启下的重要一环。所有 Android 应用程序都运行在 Dalvik VM 之上，如果 Android 应用程序需要调用核心库中的库函数，Dalvik VM 将调用本地接口(Native Interface)并执行核心库中的函数。在 Dalvik VM 之上运行的程序或应用是跨平台的，但 Dalvik VM 是和操作系统及硬件相关的。其依赖操作系统掌管的一些功能，如线程调度、内存管理等。和操作系统与底层硬件相关一样，Dalvik VM 的解释器和 JIT 都因硬件的不同而有不同的实现，如 ARM 系列、MIPS 以及 Intel X86 等平台都对应有不同的实现。

作为 Android 平台至关重要的中间件，Dalvik VM 的输入是经过 dx 工具打包好的 Dex 文件，输出是程序执行结果。图 1.2 以 Hello. java 为例，给出了一个普通应用程序在 Dalvik 虚拟机中的执行流程。

Google 推荐 Android 应用程序使用的编程语言是 Java。如果编写性能要求高的程序，也可使用 C/C++ 。Dalvik VM 在 API 上和 Oracle 公司的 Java API 是兼容的，减少了应用程序开发难度。编写好的 Java 程序可以直接用 PC 上的 Java 编译器编译为 class 文件。

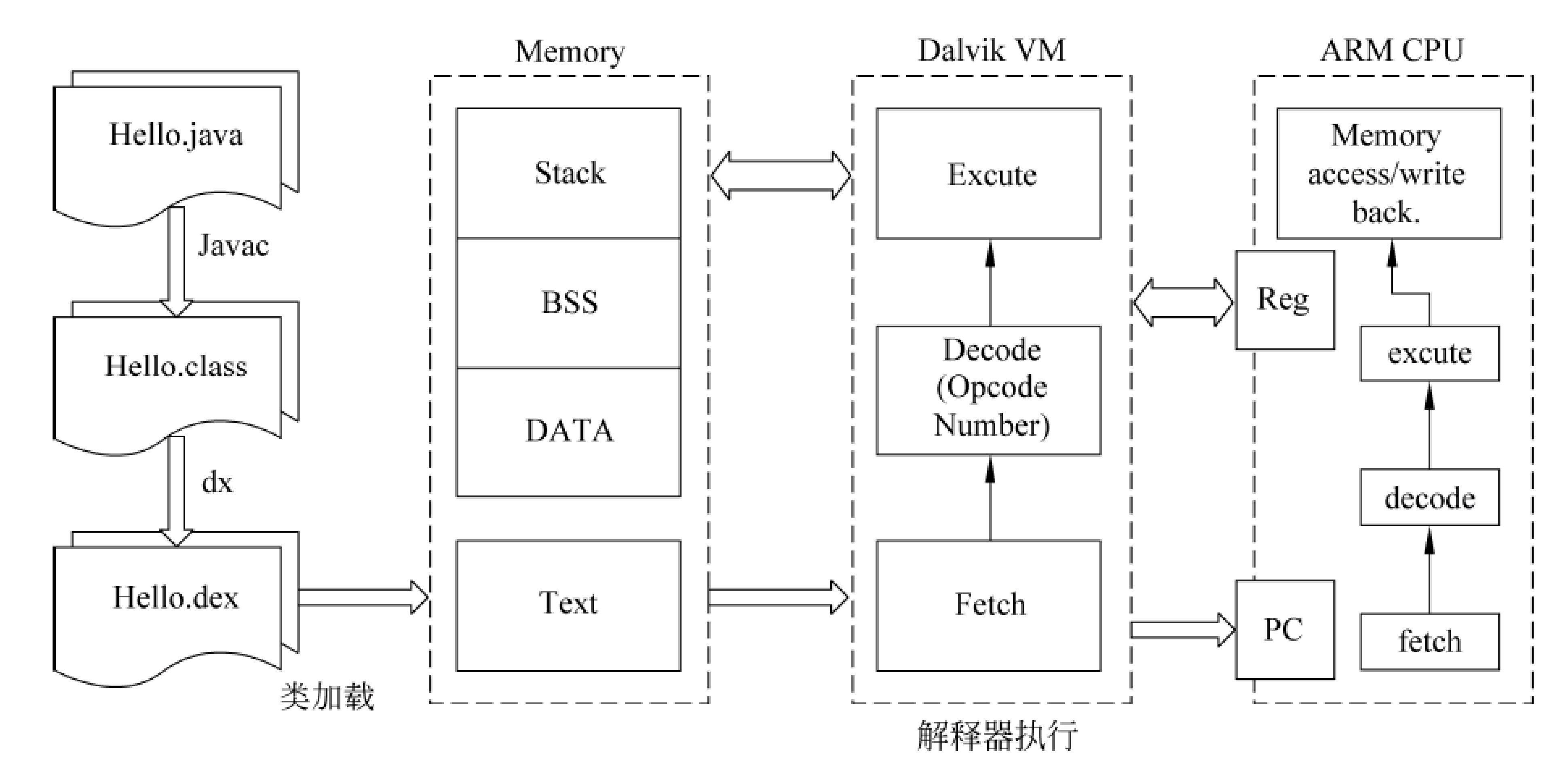


图 1.2　Dalvik VM 执行应用程序流程图

之后，需要使用 Dalvik VM 提供的 dx 工具对其进行转换。Dalvik VM 最初是基于 ARM 这个 RISC 架构设计的，和传统的基于栈的 Java 虚拟机不同，其是基于寄存器的。基于寄存器的虚拟机相比于基于栈的虚拟机，有更少的 losd/store 之类的访存类指令，也就是更少的内存访问和指令分派次数。根据统计，相比于 JVM，在 Dalvik VM 上运行的应用程序要少 47%的指令数。dx 工具解析 class 文件，合并多个 class 文件，转换为基于寄存器的字节码，并优化字节码，最终生成 Dex 文件。生成的 Dex 文件将作为 Dalvik VM 的输入。

Dalvik VM 启动并初始化后，Dex 文件将被映射到内存区，解释器开始将 Dex 文件中的每一条字节码解释为本地代码并运行。如图 1.2 所示，解释器的工作流程和真实 CPU 的工作原理非常相像，都包括取指、解码以及执行。具体的实现原理较为简单，以一个循环来完成一条字节码的解释工作。Dalvik 解释器从内存中取得字节码，并对字节码进行解码（获取字节码号），之后跳转到对应的代码段执行。无论是 C 语言编写的解释器还是汇编语言编写的解释器，每一条字节码都有一段与之等价的最终执行。如果有 JIT 的支持，JIT 将编译热代码，并将编译后的 Native Code 安装至内存区。再次执行时，解释器将跳转至相应 Native Code 执行，这将大幅度提高执行速度。

点拨　Dalvik 虚拟机的设计者 Dan Bornstein 的个人主页是 http://www.milk.com/home/danfuzz/。Dalvik 虚拟机源码目录下的 docs 目录中也有对虚拟机进行相关介绍的文档，可予以参考。

1.1.2　Dalvik 虚拟机的功能

1.1.1 节中已经简要介绍了 Dalvik 虚拟机的功能，概括来说，Dalvik 虚拟机主要完成对象生命周期的管理、堆栈的管理、线程管理、安全和异常的管理以及垃圾回收等重要功能。在 Dalvik 设计的过程中充分利用了 Linux 进程管理的特点，使其可以同时运行多个进程，这就使得在 Android 系统上可以同时运行多个应用程序，每一个应用程序都对应后台一个独立的虚拟机进程。

Dalvik 虚拟机考虑到运行环境资源相对紧张的特点，对线程管理、类加载、内存管理、

本地接口、反射机制、解释器、即时编译等主要功能模块做了相应的优化及创新。具体功能如下。

进程管理：进程隔离和线程管理，每一个 Android 应用在底层都会对应一个独立的 Dalvik 虚拟机实例，所有的 Android 应用的线程都对应一个 Linux 线程，进程管理依赖于 Zygote 机制实现。

Zygote 线程管理：每一个 Android 应用都运行在一个 Dalvik 虚拟机实例里，而每一个虚拟机实例都是一个独立的进程空间，这样做的优点是最大程度地保护了应用的安全和独立运行。Zygote 进程是在系统启动时产生的，它会完成虚拟机的初始化、库的加载、预置类库的加载和初始化等操作，而在系统需要一个新的虚拟机实例时，Zygote 通过复制自身，最快速地提供一个虚拟机实例。另外，对于一些只读的系统库，所有虚拟机实例都和 Zygote 共享一块内存区域，极大地节省了内存开销。Zygote 是虚拟机实例的孵化器。它通过 init 进程来启动，首先会孵化出 System_Server 启动系统服务，并且监听 Socket 等待请求命令，当有一个应用程序启动时，Zygote 会调用相应函数 fork 出一个新的进程来执行应用程序。Zygote 进行 fork 时有以下三种不同的方式。

(1) fork()，fork 一个普通的进程，该进程属于 Zygote 进程。

(2) forkAndSpecialize()，fork 一个特殊的进程，该进程不再是 Zygote 进程。

(3) forkSystemServer()，fork 一个系统服务进程。

类加载：解析 Dex 文件并加载 Dalvik 字节码。

对 Android 源码经过 Android SDK 编译后生成的 APK 文件进行一系列的处理，再在展开的 APK 文件中找到 classes.dex 文件并从 Dex 文件中加载 Dalvik 字节码供虚拟机执行模块调用。

类加载器在虚拟机中负责查找并加载字节码文件，即通过提取二进制的字节码文件，并将其存入该类的运行时数据结构，供解释器执行。在加载目标类时，还需要将该类的所有超类和超类实现的接口，需要加载的类分为基础类库和用户自定义类。当虚拟机装载某个类型时，类加载器会定位相应的字节码文件，然后读入这个字节码文件，提取其中的数据信息，并将这些信息存储到对应的内存中。

Android 系统启动时，类加载器会加载所有基础类库，用户自定义类是在虚拟机运行时才载入的。当虚拟机在运行时需要调用的一个成员方法或者一个成员变量所属的类没有被解析的时候，虚拟机会调用类加载模块，对这个类、超类以及这些类的相关接口进行加载和连接。

内存管理：分配系统启动初始化和应用程序运行时需要的内存资源。

Dalvik 虚拟机内存管理分为内存分配和垃圾回收。内存分配的底层依赖是基于 Doug Lea 编写的 dlmalloc 内存分配器，在 Heap 上完成，按照分配规则，每分配一个内存区域经过数次尝试。如果分配不成功，就启动垃圾收集按照相应策略进行垃圾收集。Dalvik 虚拟机在垃圾回收时使用 Mark Sweep 算法，该算法一般分为 Mark 阶段和 Sweep 阶段。Mark 阶段就是标记出活动对象，使用栈来保存根集合，然后对栈中的每一个元素，递归追踪所有可访问的对象，对于所有可访问的对象，在 markBits 位图中该将对象的内存起始地址对应的位设为 1。这样当栈为空时，markBits 位图就是所有可访问的对象集合。垃圾收集的第二步就是回收内存，在 Mark 阶段通过 markBits 位图可以得到所有可访问的对象集合，而

liveBits 位图表示所有已经分配的对象集合。因此通过比较这两个位图，liveBits 位图和 markBits 位图的差异就是所有可回收的对象集合。

本地接口：让既有代码继续发挥作用。在 Java 代码中调用其他代码的接口。

JNI 是 Java Native Interface 的缩写，即 Java 本地调用。从 Java 1.1 开始，Java Native Interface(JNI)标准成为 Java 平台的一部分，它允许 Java 代码和其他语言写的代码进行交互。JNI 一开始是为了本地已编译语言，尤其是 C 和 C++ 而设计的，但是它并不妨碍使用其他语言，只要调用约定受支持就可以了。Android 系统的 Dalvik 虚拟机实现了这套接口，供 Dalvik 虚拟机的 Java 应用与本地代码实现互相调用。

注重处理速度：和本地代码(C/C++ 等)相比，Java 代码的执行速度相对慢一些。如果对某段程序的执行速度有较高的要求，建议使用 C/C++ 编写代码。而后在 Java 中通过 JNI 调用基于 C/C++ 编写的部分，常常能够获得更快的运行速度。例如，图形处理等需要大量计算的情况下通常会采用该机制。

直接进行硬件控制：为了更好地控制硬件，硬件控制代码通常使用 C 语言编写。而后借助 JNI 将其与 Java 层连接起来，从而实现对硬件的控制。Dalvik 虚拟机使用一些本地代码编写的已编译的代码库与硬件、操作系统直接进行交互。

对既有本地代码的复用：在程序编写过程中，常常会使用一些已经编写好的本地代码(如 C/C++ 代码)，既提高了编程效率，又确保了程序的安全性与健壮性。在复用这些本地代码时，就要通过 JNI 本地调用接口来实现。

从整体来看，使用 JNI 主要在于可以直接重用一些数量庞大的本地代码，对于那些性能要求比较高的代码而言，通过 JNI 机制使用本地代码可以增强程序的性能，减少功耗，特别是对于内存较小、CPU 运算速度有限和电池电量不够充沛的嵌入式移动手机设备而言，提高性能减少功耗这一点就显得尤为重要。然而，使用 JNI 调用接口也会带来一些负面影响，比如有些时候失去了平台的可移植性，但是从综合角度考虑，在有些情况下，JNI 机制带来的优点要大于它的缺点。

反射机制：能动态查看、调用、更改任意类中的方法和属性，并能根据自身行为的状态和结果，调整或修改应用所描述行为的状态和相关的语义。

反射机制是 Dalvik 虚拟机中的核心机制之一，也算作一类工具，合理地使用反射机制能使 Java 代码变得更加简洁、灵活。同时，反射机制是 Java 被当作准动态语言的一个关键性质。反射机制允许程序在运行的过程中通过反射机制的 API 取得任何一个已知名称的类的内部信息，包括其中的描述符、超类，也包括属性和方法等所有信息，并且可以在程序运行时改变属性的相关内容或调用其内部的方法。反射机制在实现其功能时首先通过上层应用 API 运用 JNI 本地调用机制调用本地方法集中的函数，再向下层调用 Dalvik 虚拟机中的内部函数，最后将结果逐层返回到最上层的应用。

解释器：根据自身的指令集 Dalvik ByteCode 解释字节码。

解释器是 Dalvik 虚拟机的执行引擎，它负责解释执行 Dex 字节码。在 Android 4.04 版本的虚拟机中，解释器共有两种实现，分别是 C 语言实现和汇编语言的实现，分别称作可移植型(Portable)解释器和快速型(Fast)解释器。在字节码加载已经完毕后，Dalvik 虚拟机解释器开始取指解释字节码。解释器根据指令进行相应的操作，然后返回相应的结果。

在 mterp 目录下有一个重要的部分，即 out 目录，存储的是针对各个平台的解释器程序

和C语言实现的通用解释器。在解释器执行时，仿照真实机器执行，分别有取指、执行过程。在每个操作码的解释程序完成后，就取下一条指令，并跳转执行，以提高效率。解释器的入口代码位于 interp 目录下的 Interp.cpp 中的 dvmInterpret 函数，在该函数中，根据系统参数的不同，会分别选择 Fast 解释器和 Portable 解释器来执行 dvmMterpStd 和 dvmInterpretPortable。

执行引擎是虚拟机的核心，负责执行字节码或者本地方法。Dalvik 虚拟机主要采用解释执行的方式，Android 4.04 版本的虚拟机同时支持 JIT 编译器技术。

解释执行方式是指虚拟机在执行过程中将每一条字节码指令解释成本地代码运行，其工作原理比较简单，字节码的解释过程是一个循环结构，每次循环完成一条字节码的执行工作：取指令、执行功能和跳转。简言之，解释执行方式就是把字节码指令的功能用特定平台上的语言来实现。解释执行是利用该平台的资源，不需要进行附加的硬件设计，利用软件来实现虚拟机的方式。

即时编译：将反复执行的热代码编译成本地码，降低解释器压力。

JIT 技术是将字节码编译成本地代码执行，当某一个方法第一次被调用时，JIT 编译器将对虚拟机方法表所指向的字节码进行编译，编译后表中的指针将指向编译生成的机器码，如果程序再次执行该方法时，将执行经过编译的代码，提高了执行速度。

众所周知，程序执行有两种方式，分别为解释和编译。解释方式是在逐句读取源程序逐句翻译成机器码再执行；而编译方式则是在运行程序前，整个翻译为等价的目标程序，计算机直接执行该目标程序。JIT(Just-In-Time)，中文含义为即时编译，又称为动态编译，执行时动态地编译程序，以缓解解释器的低效工作。JIT 混合了两种技术，解释器解释时，编译部分程序，并在下次直接执行该编译后的源程序。

对于 Java 这类语言来说，不论是解释器还是 JIT 模块，都是在中间代码基础上做文章。Java 语言编译后，生成和平台无关的中间代码。不同平台上的虚拟机(JVM)都可以执行相同的 Java 中间代码，同时需要处理和操作系统和硬件平台相关的部分。这也是大多数解释型语言采用的模型。虚拟机中最主要的是解释器，负责解释执行中间代码。

虽有跨平台的优势，但同时也带来了运行效率低的直观感受。究其原因，是因为其执行原理是一句一句翻译字节码。比如，字节码中出现循环，根据解释器的原理，它需要重复地解释并执行这一组程序。这使得纯粹基于解释器运行的程序效率十分低下，造成了浪费。

JIT 是解决这个缺陷的一种有效手段。通过将字节码编译为 Native Code，让解释器不再重复执行这些热点代码片段。而且，相比于解释器，JIT 编译器可以更高效地利用 CPU 和寄存器。同时在编译的过程中，可以进行部分低级代码优化，比如常数传播、取消范围检查、复制传播等。尽可能地生成媲美编译器编译的二进制代码。之后执行编译生成的 Native Code，从而达到加速执行应用程序的目的。

1.1.3 Dalvik 虚拟机与 Java 虚拟机的区别

Dalvik 虚拟机并不是按照 Java 虚拟机的规范来实现的，二者并不兼容；二者的最大差异是架构上的差异，即 Dalvik 虚拟机的设计是基于寄存器的，Java 虚拟机的设计是基于栈的。

那为何 Java 虚拟机选择基于栈的设计，而 Dalvik 虚拟机选择基于寄存器的设计呢？这和 Dalvik 运行的硬件环境有关。众所周知，Dalvik 运行于手持设备之上，手持设备大多采用的是 ARM 或是相似的 RISC 结构的 CPU，这些硬件有个特点，相对来说，RISC 架构有更丰富的寄存器。如果 Dalvik 的字节码也采用 RISC 架构设计，在中间语言这一层，就有丰富的寄存器。这就使得在解释时，避免了过多的访存指令。

Dalvik 虚拟机和 Java 虚拟机另外一个显著的区别是两个虚拟机上运行的文件格式不同，前者具有自己专有的文件格式(.dex)，后者则是字节码文件(.class)。在 Java 程序中，Java 类会被编译成一个或多个字节码文件，然后打包为.jar 文件，Java 虚拟机从相应的.class 和.jar 文件中获取相应的字节码。Android 应用程序也是用 Java 语言编写的，但在编译成.class 文件后，还会通过 dx 工具将所有.class 文件统一封装成一个.dex 文件，Dalvik 虚拟机从中读取指令和数据。

点拨　掌握 Dalvik 虚拟机与 Java 虚拟机的联系和区别对于学习 Dalvik 虚拟机是非常有帮助的，目前除源码目录中的一小部分技术文档外，官方并没有给出关于 Dalvik 虚拟机的技术文档，建议在学习 Dalvik 虚拟机前类比 Java 虚拟机来学习，相信可以达到事半功倍的效果。其官方(The Java® Virtual Machine specification)的网址为 http://docs.oracle.com/javase/specs/jvms/se7/html/index.html。

1.1.4　Dalvik 虚拟机的特性

Dalvik 虚拟机非常适合在移动终端上使用，相对于在桌面系统和服务器系统运行的虚拟机而言，它不需要很快的 CPU 速度和大量的内存空间。根据 Google 的测算，64MB 的 RAM 已经能够令系统正常运转了。其中 24MB 被用于底层系统的初始化和启动，另外 20MB 被用于启动高层服务。当然，随着系统服务的增多和应用功能的扩展，其所消耗的内存也势必越来越大。

Android 是由 Google 公司基于移动设备而开发的嵌入式系统，具有优良的性能表现以及较低的硬件配置需求，因此使其迅速成为目前移动终端之上的主流操作系统。这种优势的体现主要得益于 Google 对作为 Android 系统基石的 Dalvik 虚拟机所做出的大量优化。实际上，Dalvik 虚拟机并不是一个标准的 Java 虚拟机，因为它不符合 Java 虚拟机设计规范。Dalvik 虚拟机是一个针对嵌入式系统中低速 CPU 和内存受限等特点，经过专门设计优化而实现的 Java 语言虚拟机。

Dalvik 虚拟机是基于寄存器架构的，相比基于堆栈的标准 Java 虚拟机，基于寄存器设计的 Dalvik 虚拟机的操作指令更长，因此用于实现相同程序的指令数则更少，同时对于较大的程序，其花费的编译时间也更少。研究表明，寄存器架构虚拟机比起堆栈虚拟机，相同功能程序字节码减少了 47%；代码总量增加 25%；内存访问次数减少 37.42%；程序整体执行时间减少 32.30%。Dalvik 虚拟机使用专用的 ByteCode 字节码，在 Android 2.1 中，共有 218 个字节码，分布在 Ox00 与 Oxff 之间，中间保留了 38 个无效字节码。字节码的存储方式为 16 位比特的无符号数。同时，为了能在资源受限的嵌入式系统中实现更高的性能，Dalvik 虚拟机使用专门定制的 Dex 文件作为可执行文件。Dex 文件是对多个 Class 文件进行高效整合的产物，使各个类共享相同的数据，减少了冗余性，结构十分紧凑。

1.2　Ubuntu Linux 系统安装

在系统安装前，需要获取一个版本为 Ubuntu 10.04 的系统镜像，该镜像大小为 695MB，目前网络上的资源很多且下载速度较快，读者自行下载即可。在获取了安装程序之后，需要根据需求选择安装方式。

安装方式主要分为两种：

① 在本机上直接安装 Ubuntu 系统；

② 在虚拟机中安装 Ubuntu 系统。

两种方式各有利弊，简单来说，在本机上直接安装会较少地占用系统资源，系统的工作效率较高，适合机器配置较低的用户，但对于不常用 Linux 系统的用户来说可能会有较多的不便；在虚拟机中安装相对来说更加安全且简便，尤其对于长期使用 Windows 的用户来说，不会干扰其习惯的工作模式，但在虚拟机中运行 Ubuntu 系统对机器的性能要求较高，建议机器配置较高的用户选择这种安装方式。

点拨　Ubuntu 10.04 的硬件门槛很低，但为了保证学习的顺利，建议读者参照如下标准配置设备。①处理器：尽量选用 Inter I 系列 64 位多核处理器，或者其他支持 64 位虚拟化技术的处理器。②内存：4GB 以上。③硬盘：至少 60GB。④图形性能不做要求。

下面通过图示简单介绍一下系统的安装流程。

(1) 将烧录好的系统盘插入计算机，并启动计算机，正确读取系统盘后会出现如图 1.3 所示的界面。

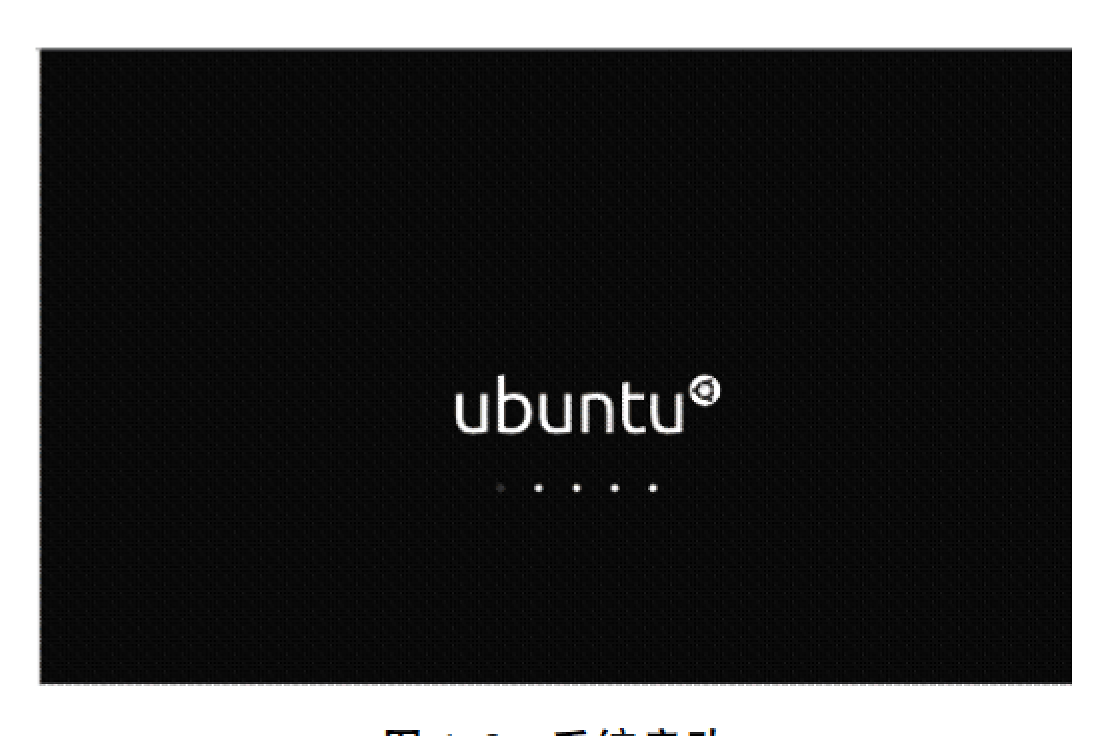


图 1.3　系统启动

这里需要指出，选择直接安装的用户可以将系统镜像烧录到 DVD 光盘或是 U 盘中，以制成系统启动盘。关于启动盘的制作方法，网上有大量的资料，在此就不再赘述；选择虚拟机安装的用户可以直接加载系统镜像，其后安装步骤和直接安装完全相同。

(2) 经过一段时间的等待，就正式开始系统的安装了。首先，需要对系统语言、键盘布局以及时区等基本信息进行设定，如图 1.4 所示。

(3) 完成相关的设定之后，将要为目标系统选择相应的安装位置，可以直接选择将系统安装到一块硬盘中，也可以通过分区工具对目标硬盘进行分区，并将系统装入指定分区中，如图 1.5 所示。



图 1.4 语言、键盘及时区设置



图 1.5 系统分区设置

(4) 在这个界面中，需要对用户名、计算机名以及密码等信息进行设定。值得注意的是，密码一定要认真记录，在 Linux 操作系统中经常需要输入密码以获取操作权限，如图 1.6 所示。

(5) 在完成用户名、密码等信息设定之后，系统将开始自动安装，如图 1.7 所示。

(6) 系统安装结束后会提示将重新启动，在启动完毕后正确输入之前设定的用户名以及密码则可以登录到系统，如图 1.8 所示。

至此，Ubuntu 系统的安装就简要介绍完毕。这也完成了 Android 源码调试分析的第一步工作。



图 1.6 创建账户



图 1.7 系统安装



图 1.8 系统登录

1.3 工作目录设置

工作位置的选择有一定的自主性，可以依据个人习惯设置相应的工作目录，在本书中作者将源码的存储位置设置为主文件夹下的 Android 4.0.4 文件夹中。实际上创建一个空的文件的方式有很多，但在本书中一律使用“终端”进行操作，因此读者有必要在闲暇时了解 Linux 系统命令的使用。

首先，开启一个终端，操作如图 1.9 所示。



图 1.9 开启终端

开启终端后，即可输入相关的 Linux 命令并回车以完成相应的操作，如图 1.10 所示。

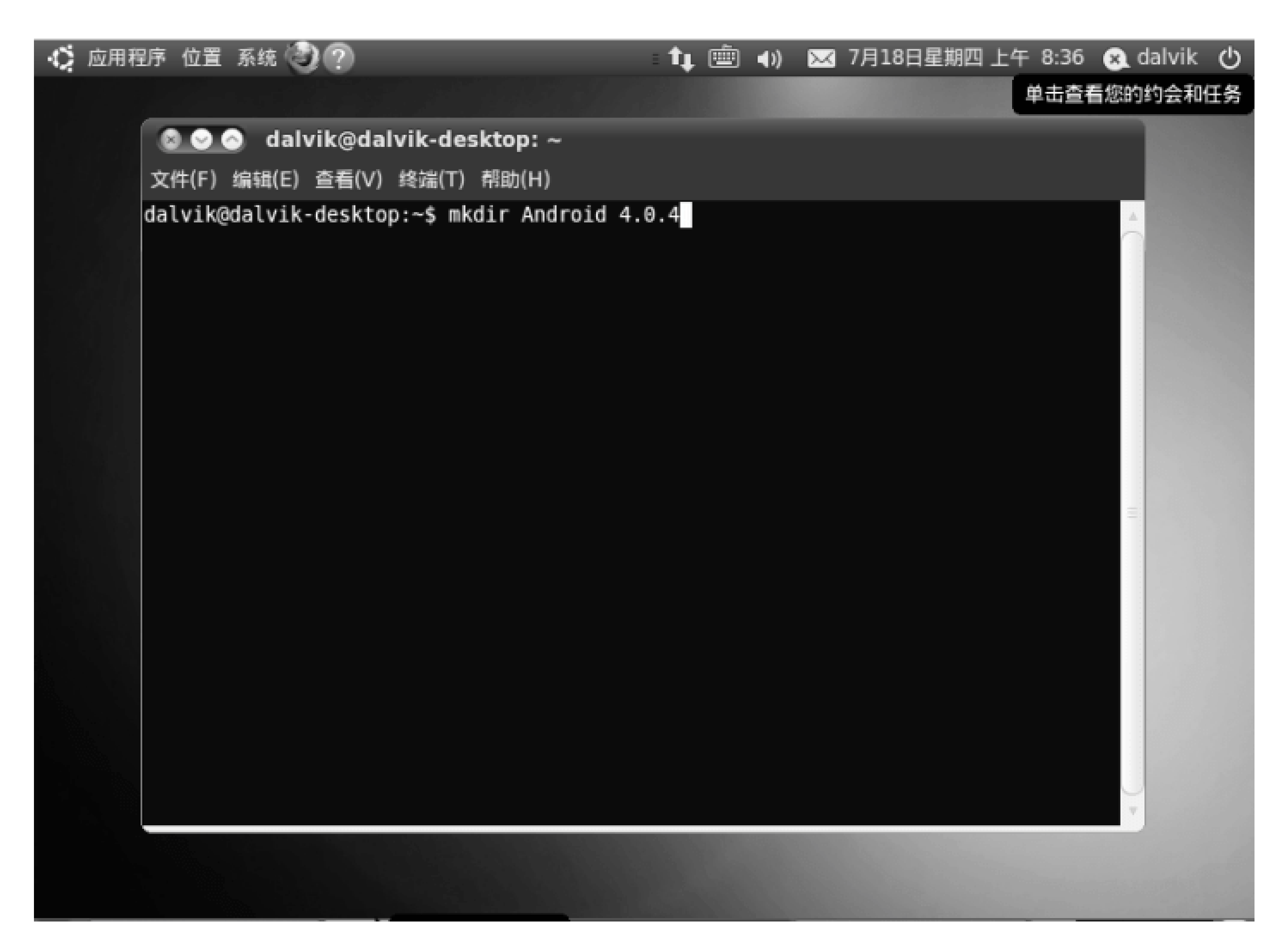


图 1.10 创建工作目录

本部分实际上只需要一条命令，即 mkdir Android 4.0.4，用于在主目录下创建一个名为 Android 4.0.4 的空文件夹，该文件夹是用来存储 Android 系统源码，也是工作的主要位置之一。

1.4　下载、编译和运行 Android 内核源代码

1.4.1　下载 Android 内核源代码

完成了工作目录的设定后，就可以开始准备下载源码了。本书将对下载源码过程中的每一步操作进行介绍，力求帮助读者理解这些操作的真正意义，而不是单纯地输入指令并执行。

第 1 步　安装 Git

Git 是用于 Linux 内核开发的版本控制工具。与 CVS、Subversion 一类的集中式版本控制工具不同，它采用了分布式版本库的作法，不需要服务器端软件，就可以运作版本控制，使得源代码的发布和交流极其方便。简单来说，在本书中 Git 就是用来管理下载 Android 系统源码的。

输入命令：sudo apt-get install git-core，并回车。随后系统会要求输入用户密码以取得安装权限，在正确输入密码后系统将自动安装目标程序，如图 1.11 所示。

```
dalvik@dalvik-laptop:~$ sudo apt-get install git-core
[sudo] password for dalvik:
```

图 1.11　输入安装命令及系统密码

系统将提示是否继续，输入 Y 后继续安装，如图 1.12 所示。

```
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
将会安装下列额外的软件包：
  libdigest-sha1-perl liberror-perl patch
建议安装的软件包：
  git-doc git-arch git-cvs git-svn git-email git-daemon-run git-gui gitk
  gitweb diffutils-doc
下列【新】软件包将被安装：
  git-core libdigest-sha1-perl liberror-perl patch
升级了 0 个软件包，新安装了 4 个软件包，要卸载 0 个软件包，有 317 个软件包未被升
级。
需要下载 5,811kB 的软件包。
解压缩后会消耗掉 12.1MB 的额外空间。
您希望继续执行吗？[Y/n]y
```

图 1.12　提示是否继续安装

值得注意的是，在安装过程中需要保持网络连接正常，因为系统需要对目标程序的安装文件进行下载。

出现如图 1.13 所示的信息后，即表示安装结束。

第 2 步　安装 curl

curl 是利用 URL 语法在命令行方式下工作的文件传输工具，其支持多种网络协议，如 FTP，FTPS，HTTP，HTTPS，GOPHER，TELNET，DICT，FILE 以及 LDAP 等。简单来

```
获取：1 http://cn.archive.ubuntu.com/ubuntu/ lucid/main liberror-perl 0.17-1 [23
.8kB]
获取：2 http://cn.archive.ubuntu.com/ubuntu/ lucid/main libdigest-sha1-perl 2.12
-1build1 [26.2kB]
获取：3 http://cn.archive.ubuntu.com/ubuntu/ lucid-updates/main git-core 1:1.7.0
.4-1ubuntu0.2 [5,638kB]
获取：4 http://cn.archive.ubuntu.com/ubuntu/ lucid/main patch 2.6-2ubuntu1 [123k
B]
下载 5,811kB, 耗时 23秒 (246kB/s)
选中了曾被取消选择的软件包 liberror-perl。
(正在读取数据库 ... 系统当前总共安装有 125731 个文件和目录。)
正在解压缩 liberror-perl (从 .../liberror-perl_0.17-1_all.deb) ...
选中了曾被取消选择的软件包 libdigest-sha1-perl。
正在解压缩 libdigest-sha1-perl (从 .../libdigest-sha1-perl_2.12-1build1_i386.deb
) ...
选中了曾被取消选择的软件包 git-core。
正在解压缩 git-core (从 .../git-core_1%3a1.7.0.4-1ubuntu0.2_i386.deb) ...
选中了曾被取消选择的软件包 patch。
正在解压缩 patch (从 .../patch_2.6-2ubuntu1_i386.deb) ...
正在处理用于 man-db 的触发器...
正在设置 liberror-perl (0.17-1) ...
正在设置 libdigest-sha1-perl (2.12-1build1) ...
正在设置 git-core (1:1.7.0.4-1ubuntu0.2) ...
正在设置 patch (2.6-2ubuntu1) ...
```

图 1.13　安装过程截图

说，通过给定一个目标资源的 URL，curl 工具将该 URL 的目标资源下载至本地。

输入命令：sudo apt-get install git-core curl，并回车。curl 的安装过程和前面安装 Git 的过程类似，需要保持网络连接，参见图 1.14。

```
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
git-core 已经是最新的版本了。
下列【新】软件包将被安装：
  curl
升级了 0 个软件包，新安装了 1 个软件包，要卸载 0 个软件包，有 317 个软件包未被升级。
需要下载 209kB 的软件包。
解压缩后会消耗掉 324kB 的额外空间。
获取：1 http://cn.archive.ubuntu.com/ubuntu/ lucid-updates/main curl 7.19.7-1ubuntu1.3 [209kB]
下载 209kB, 耗时 1秒 (115kB/s)
选中了曾被取消选择的软件包 curl。
(正在读取数据库 ... 系统当前总共安装有 126266 个文件和目录。)
正在解压缩 curl (从 .../curl_7.19.7-1ubuntu1.3_i386.deb) ...
正在处理用于 man-db 的触发器...
正在设置 curl (7.19.7-1ubuntu1.3) ...
```

图 1.14　安装 curl 工具

出现如上信息，即表示安装结束。

第 3 步　获取 repo，通过 curl 下载 repo 文件

repo 文件实际上是 Google 的开发人员所编写的一个 Python 脚本文件，其作用是通过调用 Git 来实现下载、管理 Android 项目的软件仓库。因此，需要通过 curl 获取该 repo 文件，其方法如下。

输入命令：curl https://dl-ssl.google.com/dl/googlesource/git-repo/repo＞～/bin/repo，并回车，如果命令被正确执行后，应该可以在 bin 文件夹下找到一个名为 repo 的文件，如图 1.15 所示。

注：如果系统提示找不到 bin 文件夹，可以在主目录下创建名为 bin 的文件夹，然后再运行该命令，如图 1.16 所示。



图 1.15 repo 文件下载至 bin 文件夹中



图 1.16 在根目录下创建 bin 文件夹

第 4 步 修改执行权限，取得修改 repo 文件的权限

刚刚下载下来的 repo 文件是不能直接使用的，需要修改其中一些参数的值。另外，由于 repo 文件可能是一个只读文件，因此通过下面的操作获取它的读写权限。

输入命令：chmod a+x ~/bin/repo，并回车。

第 5 步 修改相关的环境变量

需要将 bin 文件和路径添加到环境变量中，操作如下。

输入命令：gedit ~/. bashrc，如图 1.17 所示。

```
dalvik@dalvik-laptop:~$ gedit ~/.bashrc
```

图 1.17 编辑环境变量配置文件

随后在文件的最后添加：export PATH=$PATH:~/bin 语句并回车，如图 1.18 所示。

```
# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if [ -f /etc/bash_completion ] && ! shopt -oq posix; then
    . /etc/bash_completion
fi
export PATH=$PATH:~/bin
```

图 1.18 增加环境变量

第 6 步 设置相关的下载信息

首先要通过使用 cd 命令进入前面设定的工作目录，这样做的目的是表示以后所有的操作都在该目录下进行，这是 Linux 系统的一种使用操作逻辑，不太清楚其中道理的读者，可以阅读一下相关的资料，对 Linux 系统操作进行学习；随后，需要设置相应的下载信息，主要包括目标资源的 URL 以及目标的 Android 源码版本，在本书中所用到的版本为 android-4.0.4_r2.1。

输入命令：cd Android 4.0.4，并回车，进入目标文件夹，此步骤比较关键，如果弄错了文件夹，则源码也将被下载至错误的文件夹中。

输入命令：repo init -u https://android.googlesource.com/platform/manifest -b android-4.0.4_r2.1，并回车，完成下载信息的设置。

第 7 步 下载 Android 源码

完成了下载信息的设定后，即可开始下载指定源码，其操作如下。

输入命令：repo sync，并回车。源码大小为 11GB，其下载时间较长，需要耐心等待。

注：如果下载中中断，重新执行 repo sync 命令即可。

通过以上 7 个步骤的操作并且系统执行无误的情况下，Android 4.0.4 的源码将被完整地下载至本机。至此，可以准备开始对源码进行编译调试的工作。

1.4.2 整体编译 Android 源代码

首先需要进入源码所在的文件目录下，通过使用 cd 命令即可。在读者所用环境下输入命令：cd Android 4.0.4，回车即可进入该目标目录。

随后需要设置相关的环境变量：

输入命令：source build/envsetup.sh 并回车，应当出现如图 1.19 所示的信息。

```
including device/moto/stingray/vendorsetup.sh
including device/moto/wingray/vendorsetup.sh
including device/samsung/crespo4g/vendorsetup.sh
including device/samsung/crespo/vendorsetup.sh
including device/samsung/maguro/vendorsetup.sh
including device/samsung/torospr/vendorsetup.sh
including device/samsung/toro/vendorsetup.sh
including device/samsung/tuna/vendorsetup.sh
including device/ti/panda/vendorsetup.sh
including sdk/bash_completion/adb.bash
```

图 1.19 设置环境变量图

输入命令：lunch，回车后将出现如图 1.20 所示的信息，提示选择目标编译模式。

注：如果希望在本机上以 Android 模拟器对源码进行调试，则应当选用模式：full-eng；如希望在 Samsung Galaxy Nexus 3 上对源码进行调试，则应当选用 full_maguro-userdebug 模式。在本书中，将以模式 full-eng 进行介绍。

```
dalvik@dalvik-laptop: ~/workinq/androidsource/android4.0.4$ lunch
You're building on Linux

Lunch menu... pick a combo:
     1. full-eng
     2. full_x86-eng
     3. vbox_x86-eng
     4. full_stingray-userdebug
     5. full_wingray-userdebug
     6. full_crespo4g-userdebug
     7. full_crespo-userdebug
     8. full_maguro-userdebug
     9. full_torospr-userdebug
     10. full_toro-userdebug
     11. full_tuna-userdebug
     12. full_panda-eng

Which would you like? [full-eng] █
```

图 1.20 目标编译模式选择图

输入命令：1，如果系统执行正确将会出现如图 1.21 所示的信息。

```
Which would you like? [full-eng] 1

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.0.4
TARGET_PRODUCT=full
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=IMM76L
============================================
```

图 1.21 系统执行正确提示图

下面对 Android 整体源码进行编译，还是在当前终端中输入命令：make -j4（其中参数-j 表示的是并行编译，而 4 表示的是使用 4 个 CPU 核心对源码进行编译）。随后，系统将自动开始对全部的 Android 源码进行编译，其耗时较长需要耐心等待，在 i5 处理器中大约需要 3h 左右，在虚拟机中大约需要 10h。

当编译完成后，系统将输出如图 1.22 所示的信息。

```
host C++: libdvm <= dalvik/vm/oo/Class.cpp
target thumb C++: libdvm <= dalvik/vm/oo/Class.cpp
target thumb C++: libdvm_assert <= dalvik/vm/oo/Class.cpp
target thumb C++: libdvm_interp <= dalvik/vm/oo/Class.cpp
target thumb C++: libdvm_sv <= dalvik/vm/oo/Class.cpp
host SharedLib: libdvm (out/host/linux-x86/obj/lib/libdvm.so)
target SharedLib: libdvm_interp (out/target/product/generic/obj/SHARED_LIBRARIES
/libdvm_interp_intermediates/LINKED/libdvm_interp.so)
target SharedLib: libdvm (out/target/product/generic/obj/SHARED_LIBRARIES/libdvm
_intermediates/LINKED/libdvm.so)
target SharedLib: libdvm_assert (out/target/product/generic/obj/SHARED_LIBRARIES
/libdvm_assert_intermediates/LINKED/libdvm_assert.so)
target SharedLib: libdvm_sv (out/target/product/generic/obj/SHARED_LIBRARIES/lib
dvm_sv_intermediates/LINKED/libdvm_sv.so)
target Symbolic: libdvm (out/target/product/generic/symbols/system/lib/libdvm.so
)
target Symbolic: libdvm_interp (out/target/product/generic/symbols/system/lib/li
bdvm_interp.so)
target Strip: libdvm (out/target/product/generic/obj/lib/libdvm.so)
target Symbolic: libdvm_assert (out/target/product/generic/symbols/system/lib/li
bdvm_assert.so)
target Strip: libdvm_interp (out/target/product/generic/obj/lib/libdvm_interp.so
```

图 1.22 编译输出信息

1.4.3 运行 Android 模拟器

可以通过启动模拟器来检验是否正确编译了全部的 Android 源码。

输入命令：emulator，并回车，启动模拟器。经过一段时间的等待，如模拟器成功启动，则会出现如图 1.23 所示的界面。



图 1.23 模拟器界面

该模拟器和人们常用的 Android 手机的操作模式、界面以及功能都是一样的，但不支持触屏，需要用鼠标代替手指进行操作。

至此，完成了对 Android 源码的编译工作，为分析调试 Android 源码做好了准备工作。

1.5 编译经过修改的 Android 源码

经过对源码进行分析研究，读者可能会对源码进行修改，以达到在原有 Android 系统基础上增加新的功能，提高系统执行效率等目标。但是如何对修改后的源码进行编译呢？

事实上，Android 提供了单独对个别模块进行编译的功能，即通过 mmm 命令可以实现对指定路径下的模块进行编译，例如，在终端中输入命令：mmm dalvik/vm/，即可对 Dalvik 虚拟机中的 VM 模块进行编译，但需要注意的是该指定路径下必须包含 Android.mk 文件，否则将会编译出错。根据作者在实践中所总结的经验发现，此命令的实际效果不是太好，时而会出现编译失败的情况。

除了 mmm 命令，还可以使用 make 命令，因为 make 属于一种增量编译技术，即只对修改过的源代码进行编译。因此，通过 make 命令也可以实现对那部分经过修改的 Android 源码进行编译。更重要的是，相对 mmm 命令，make 命令更加稳定、出错率低，作者推荐使用 make 命令。

1.6 开发第一个 Android 应用程序

本节将学习如何开发一个 Android 工程，本书通过一个简单的 Android 实例向读者介绍这一具体的开发流程。该程序实例实现的主要功能是在屏幕中输出“Welcome to

Android”的字符串,该程序功能虽然简单,但正所谓麻雀虽小、五脏俱全,读者可以从中学习如何通过 Eclipse 编程环境创建一个 Android 工程、编写 Android 程序代码以及进行程序的调试与运行等基本操作。

1. 新建一个 Android 工程

在编写代码之前,首先需要创建一个 Android 工程,和普通的 Java 程序一样,Android 程序的开发也是以一个工程包的形式展现的。因此,有过 Java 编程经验的读者会觉得非常熟悉,而唯一不同的是:相较于标准的 Java 应用开发,Google 为 Android 开发者提供了大量全新的 API,因此,读者应该对这些 API 进行预览式的学习,以提高程序的开发效率。Android 工程的创建流程如下。

步骤 1:在具备 Android 编程环境的 Eclipse 中,首先单击 File 菜单,随后依次选择 New Project 菜单命令,建立一个新工程。

步骤 2:在弹出的 New Project 窗口中选择 Android Project 选项,然后单击 Next 按钮,新建一个 Android 工程,如图 1.24 所示。

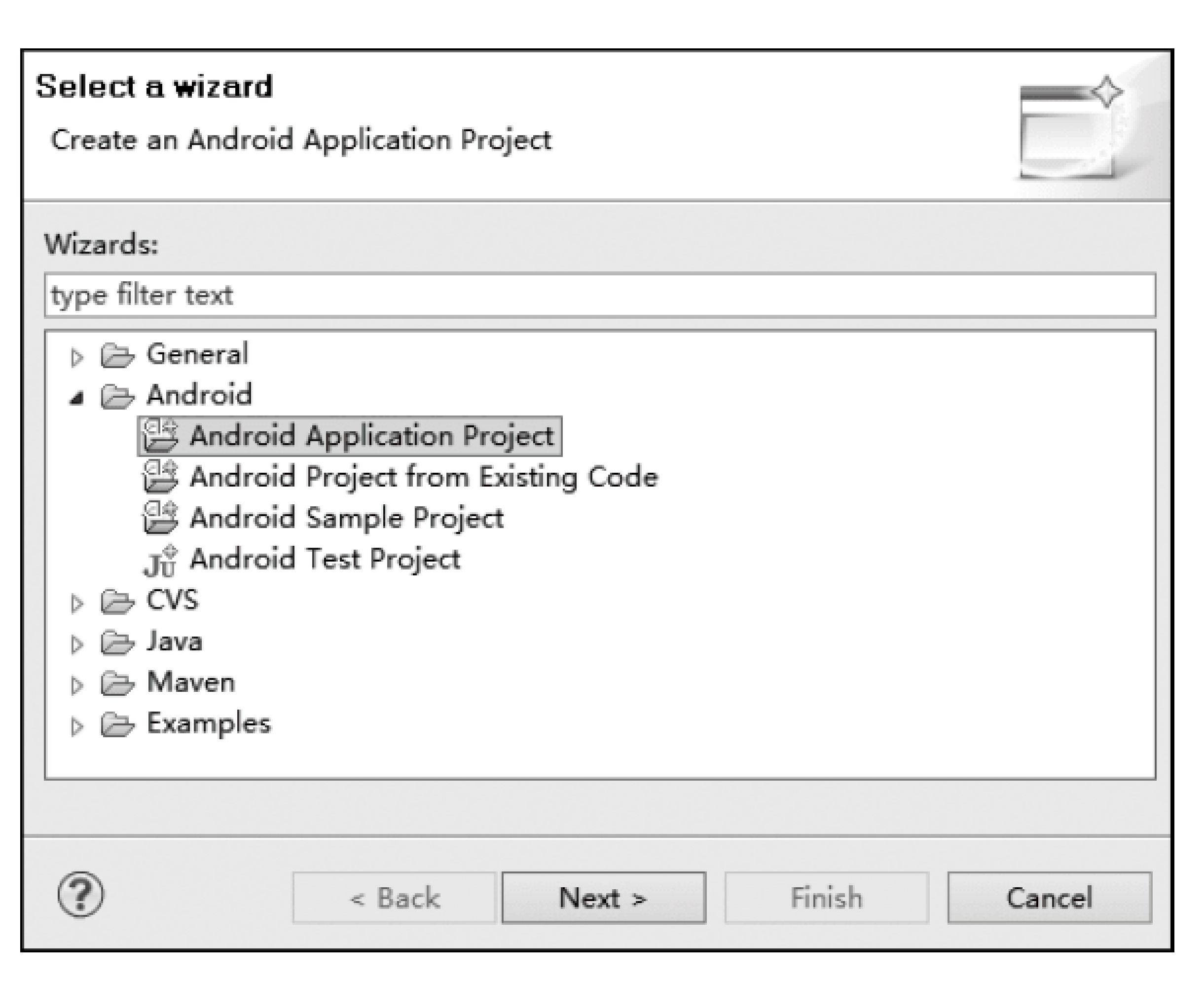


图 1.24 新建一个 Android 工程

步骤 3:在弹出的 New Android Application 窗口中设置新建的 Android 工程相关配置信息,包括应用程序名称、工程名称以及包名称等,如图 1.25 所示。

2. 编写具体代码

在完成了 Android 工程的创建工作后,便可以开始具体的代码编写工作了。在新建的工程文件中,选择并打开 Hello.java 文件,会发现系统已经自动生成了部分代码,但是这部分代码并没有实际的作用,更不会达到我们的预期目标。因此,需要对这部分代码进行改写扩充,将想要实现的效果所对应的代码加入进去。修改后的代码如图 1.26 所示。

在对原工程文件完成了上述修改之后,无法判断程序是否能够达到预期目标,因此不能将其直接编译并生成一个 Android 软件安装程序,而是要对其进行一系列的调试运行,以确保程序的正确性。

New Android Application
The application name for most apps begins with an uppercase letter
Application Name: hello
Project Name: hello
Package Name: com.example.hello
Minimum Required SDK: API 3: Android 1.5 (Cupcake)
Target SDK: API 15: Android 4.0.3 (IceCreamSandwich)
Compile With: API 15: Android 4.0.3 (IceCreamSandwich)
Theme: Holo Light with Dark Action Bar
Choose the lowest version of Android that your application will support. Lower API levels target more devices, but means fewer features are available. By targeting API 8 and later, you reach approximately 95% of the market.
< Back　Next >　Finish　Cancel

图 1.25　配置工程信息

```
package com.example.hello;

import android.os.Bundle;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        TextView message = new TextView(this);
        message.setText("Welcome to Android");
        setContentView(message);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
```

图 1.26　程序代码

3. 程序调试

Android 程序的调试方法和普通 Java 程序的区别不大，都是通过设置程序执行断点并结合后台输出信息对当前程序的执行情况进行判断，看是否能达到预期目标。因此，作者也将采用这种调试手段对该实例程序进行调试并介绍。

1) 断点设置

在 Eclipse 编程环境中，设置断点方法非常简单：只需双击目标代码行的左侧区域，当目标代码行左边出现一个蓝色小点时，即表示完成断点的设置，如图 1.27 所示。

点拨　为了方便查找故障代码位置，建议开启代码行数的显示功能。这样做的好处是在调试程序的过程中，可以根据后台抛出的异常信息快速定位到故障代码段，以达到快速修改代码的要求。开启代码行数显示功能的具体方法是：在代码左侧的空白区域中单击鼠标

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        TextView message = new TextView(this);
        message.setText("Welcome to Android");
        setContentView(message);
    }
```

图 1.27 断点设置例

右键，然后在弹出的菜单中选择 Show Line Numbers 选项。

2）程序调试

想必有些读者一定使用过 Eclipse 编写调试 Java 程序，但就调试的基本方法来讲，Android 和 Java 几乎没有不同。但在开始调试工作之前，仍然需要设置相关的调试属性。在左侧的项目文件列表中，选中项目名并单击右键，在菜单中选择 Debug as Android Application 命令，即表示当前是对一个 Android 程序进行调试，如图 1.28 所示。

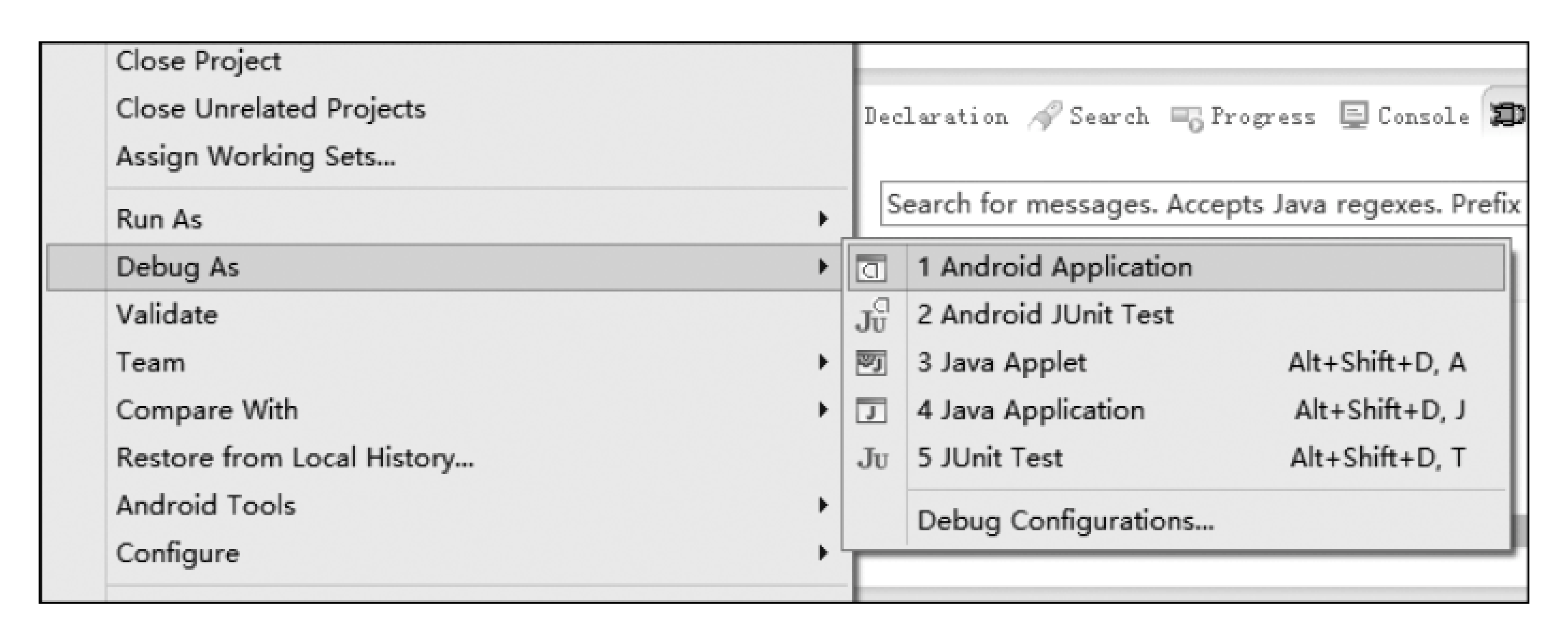


图 1.28 调试模式选择

随后选择开启 Debug 调试窗口，最终就进入了调试界面，如图 1.29 所示。

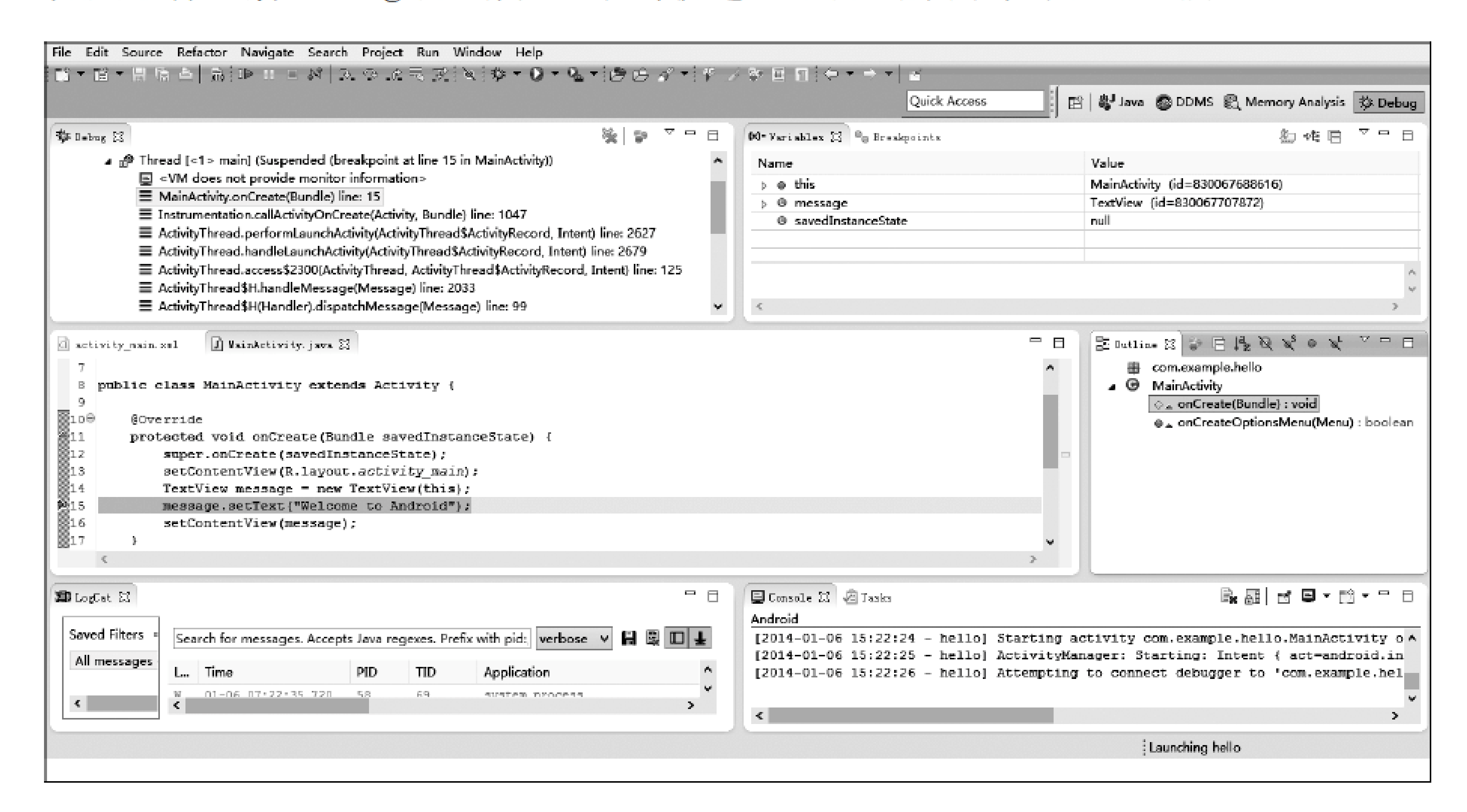


图 1.29 Debug 调试界面

在调试的过程中，可以对程序进行单步执行，当执行到断点处时，可以根据需要查看后台信息，以判断程序是否达到预期的执行效果。事实上，对程序进行调试的手段很多，很难讲孰优孰劣，因此应该根据当前情况做出判断，选用最佳的调试方法。在这里只列举了一个最常用且最基本的方法，希望读者可以多加练习，熟练掌握。

3）运行程序

如果在前面的调试阶段中程序不会再抛出异常或是错误，就可以使用模拟器对程序进行仿真运行了。操作非常简单：首先选中项目名并单击右键，在菜单中选择 Run as Android Application 命令，随后该程序将会被模拟器运行。运行情况如图 1.30 所示。

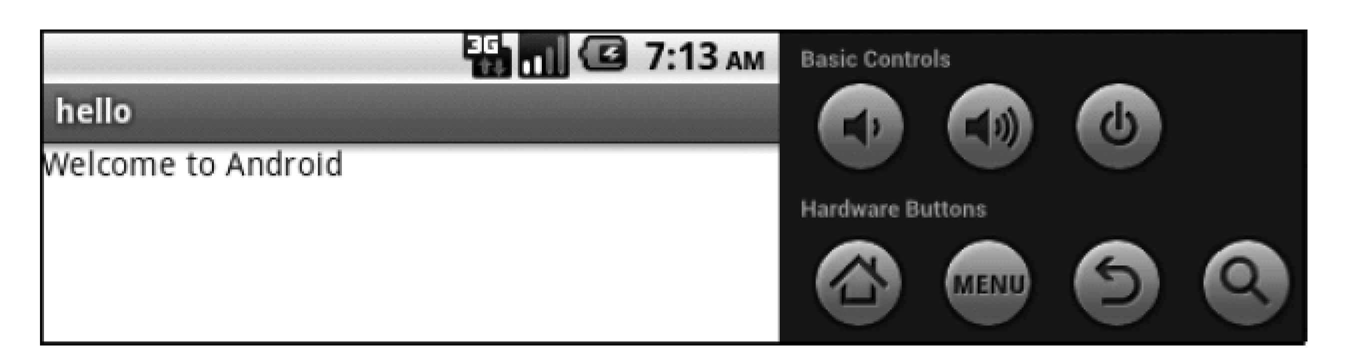


图 1.30　程序运行结果

从图 1.30 可以看到，和预想的结果相同，即在屏幕上输出“Welcome to Android”的字符串。至此，成功地编写了一个简单的 Android 程序。实际上，Android 应用程序开发还是较为容易，难就难在是否能够熟练应用 Google 所提供的大量 API，这就要求在平时开发的过程中养成勤学多练的好习惯。

小结

本章首先介绍了什么是 Dalvik 虚拟机，Dalvik 虚拟机有哪些功能，以及它和 Java 虚拟机有什么不同之处，接下来介绍了如何下载和编译 Dalvik 虚拟机源码，其中包括对于 Linux 操作系统的安装步骤。

第 2 章

源码分析辅助工具

本章主要内容

✍如何在 Linux 下用 Vim 搭建一个简易的源码阅读环境?
✍如何静态分析源码得到函数之间的关系?
✍怎样才能以文档的形式得到自己的注释?
✍如何用 GDBSERVER 动态跟踪程序执行流程?

2.1　本章概述

Android Dalvik 源码量十分巨大,因此依靠一些称手的工具将大大加快分析进度。由于采用 Ubuntu 10.04 作为编译环境,阅读和分析源码都将采用 Ubuntu 10.04 作为宿主环境。其中包括利用 Vim 及 Vim 插件搭建一个简单的源码阅读环境;利用 Doxygen 工具生成类及函数调用关系图;利用 GDBSERVER 调试跟踪程序执行流程。

2.2　Vim 源码阅读环境搭建

Vim 是 Linux 下通用的编辑器,其前身是 UNIX 下的 VI,历史十分悠久。和普通的编辑器不一样,Vim 是有模式的,这个设计会让许多初学者感到迷惑,也是其陡峭学习曲线的一部分。Vim 中最常用的模式是普通模式(Normal Mode)和插入模式(Insert Mode)。

Vim 是有模式的编辑器,分别包含普通模式和插入模式。在普通模式下,键盘的输入会被当作快捷键,而不是通常的文字录入。相应的文字录入则在插入模式下完成。在普通模式下按 i 键,则进入插入模式,此时在 Vim 的左下角应该看到“插入”或“Insert”字样。在插入模式下的使用方式和普通的记事本是一样的。当想返回普通模式时,可按 Esc 键。Vim 有一个非常强大的帮助文档,如果有些命令不知道,可以输入“:help ＜command＞”来获取 Vim 中相关命令的帮助。

普通模式下的键盘快捷键如表 2.1 所示。

表 2.1　Vim 普通模式下简单快捷键

按　键	功　能	按　键	功　能
H	光标左移一个字符	x	删除光标所在的一格字符
J	光标下移一行	:wq	保存并退出
k	光标上移一行	:!q	不保存退出
l	光标右移一个字符		

在 Ubuntu 10.04 中，默认安装的是 Vi，相比于 Vim 缺少一些功能，因此需要重新安装 Vim。Vim 的安装过程非常简单，在终端中输入以下命令，将自动完成 Vim 的安装，如图 2.1 所示。

```
dalvik@dalvik-laptop:~$ sudo apt-get install vim
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  vim-runtime
Suggested packages:
  ctags vim-doc vim-scripts
The following NEW packages will be installed:
  vim vim-runtime
0 upgraded, 2 newly installed, 0 to remove and 0 not upgraded.
Need to get 0B/7,089kB of archives.
After this operation, 27.3MB of additional disk space will be used.
Do you want to continue [Y/n]? y
```

图 2.1　Vim 安装图

安装成功后的界面如图 2.2 所示。由于网络问题或源的问题，可能会有其他错误。

```
ags.vim-tiny by vim-runtime'
Selecting previously deselected package vim.
Unpacking vim (from .../vim_2%3a7.2.330-1ubuntu3.1_amd64.deb) ...
Processing triggers for man-db ...
Setting up vim-runtime (2:7.2.330-1ubuntu3.1) ...
Processing /usr/share/vim/addons/doc

Setting up vim (2:7.2.330-1ubuntu3.1) ...
update-alternatives: using /usr/bin/vim.basic to provide /usr/bin/vim (vim) in a
uto mode.
update-alternatives: using /usr/bin/vim.basic to provide /usr/bin/vimdiff (vimdi
ff) in auto mode.
update-alternatives: using /usr/bin/vim.basic to provide /usr/bin/rvim (rvim) in
 auto mode.
update-alternatives: using /usr/bin/vim.basic to provide /usr/bin/rview (rview)
in auto mode.
update-alternatives: using /usr/bin/vim.basic to provide /usr/bin/vi (vi) in aut
o mode.
update-alternatives: using /usr/bin/vim.basic to provide /usr/bin/view (view) in
 auto mode.
update-alternatives: using /usr/bin/vim.basic to provide /usr/bin/ex (ex) in aut
o mode.

dalvik@dalvik-laptop:~$
```

图 2.2　Vim 安装成功后的界面

作为一个可扩展的编辑器，Vim 有非常强大的插件库，几乎所有插件都可以在其官网下载得到。在搭建源码阅读环境时，需要安装 ctags 和 TagList 插件。

1. ctags

说明：ctags 是 Linux 下的方便代码阅读的工具。官方解释的是产生标记文件已帮助在源文件中定位对象。通俗地说，ctags 扫描指定的源文件，找出其中包含的语法元素，并将相关的信息记录到文件中。因此，ctags 运行后，将在源码目录产生一个 tags 文件。安装过程如图 2.3 所示。

安装完成后，和 Vim 不同的是，可能会出现如图 2.4 所示的错误，不需要担心，这没什么问题。

接下来是 ctags 的配置。进入 Android 源码 dalvik 目录中，运行 ctags -R 生成 tags 文

```
dalvik@dalvik-laptop:~$ sudo apt-get install exuberant-ctags
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  exuberant-ctags
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
```

图 2.3 ctags 安装图

```
Setting up exuberant-ctags (1:5.8-2) ...
update-alternatives: error: no alternatives for ctags.
update-alternatives: using /usr/bin/ctags-exuberant to provide /usr/bin/ctags (c
tags) in auto mode.
update-alternatives: error: no alternatives for etags.
update-alternatives: using /usr/bin/ctags-exuberant to provide /usr/bin/etags (e
tags) in auto mode.
```

图 2.4 ctags 安装后的报错

件，如图 2.5 所示。在 Vim 配置文件～/. vimrc 中加入：set tags＝[Android 源码目录]/dalvik/tags。注意，tags 文件不能自动更新，如果大幅度更改源码后，需要重新生成。

```
dalvik@dalvik-laptop:~/android4.4/dalvik$ ctags -R
dalvik@dalvik-laptop:~/android4.4/dalvik$ ls
Android.mk     dexgen   dvz     libnativehelper           README.txt  unit-tests
CleanSpec.mk   dexlist  dx      MODULE_LICENSE_APACHE2    tags        vm
dalvikvm       dexopt   hit     NOTICE                    tests
dexdump        docs     libdex  opcode-gen                tools
dalvik@dalvik-laptop:~/android4.4/dalvik$
```

图 2.5 ctags 生成 tags

Vim 已经集成了对 ctags 的使用。将光标移至函数名称或结构体名上，然后按快捷键：G Ctrl＋]（按下 G，再按下 Ctrl＋]键），将会列出相关的定义处，选择相应的数字，自动跳转到相应的定义处。想要返回时，按快捷键 Ctrl＋T，自动调回原先位置。

2. TagList

说明：用过 Source Insight 的人都应该记得这么一个功能：其可以将当前文件的宏、全局变量、函数名称等显示在 Symbol 窗口，用鼠标单击就可跳转到相应的位置。Vim 中的 TagList 就实现了这样的功能。

安装：确保 home 目录下有. vim 目录，如果没有，新创建一个。然后到 Vim 官网中 http://www. vim. org/scripts/download_script. php? src_id＝19574 下载插件，解压至～/. vim 目录中。

使用：需要注意的是 Vim 必须打开文件类型自动检测功能。用 Vim 打开源码文件后，输入：Tlist，将在右侧列出函数定义等信息。

最终安装配置好 Vim 后，打开源码的界面如图 2.6 所示。右侧展示的是 dalvik/vm/Init. cpp 文件的源码，左侧是 TagList 展示的 tag，分别包括变量（variable，图 2.7）、宏（macro，图 2.8）、类（class，图 2.9）和函数名称（function，图 2.10）。

一般情况下，有这些工具就足够阅读代码了。比如想查看 tag 区域中的某个函数，可以先按 Ctrl＋W 再按 H 跳到左侧的 TagList 区域；按方向键选中要查看的函数；回车，即可跳转到这个函数。如果是想跳转到代码中的某一个函数的定义处，可以按 Ctrl＋]键，跳转到函数定义；若该函数有多个定义，将显示所有的函数定义，输入序号选中要查看的函数即可。

```
" Press <F1> to display hel|/*
                           | * Copyright (C) 2008 The Android Open Source Project
Init.cpp (/home/dalvik/andr| *
  variable                 | * Licensed under the Apache License, Version 2.0 (the "License");
    gDvm                   | * you may not use this file except in compliance with the License.
    gDvmJni                | * You may obtain a copy of the License at
    gDvmJit                | *
    gDvmICHitCount         | *      http://www.apache.org/licenses/LICENSE-2.0
                           | *
  macro                    | * Unless required by applicable law or agreed to in writing, software
    __STDC_LIMIT_MACROS    | * distributed under the License is distributed on an "AS IS" BASIS,
    kMinHeapStartSize      | * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    kMinHeapSize           | * See the License for the specific language governing permissions and
    kMaxHeapSize           | * limitations under the License.
                           | */
  class                    |
    ScopedShutdown         |/*
                           | * Dalvik initialization, shutdown, and command-line argument processing.
  function                 | */
    usage                  |#define __STDC_LIMIT_MACROS
    showJdwpHelp           |#include <stdlib.h>
    showVersion            |#include <stdio.h>
    parseMemOption         |#include <signal.h>
    handleJdwpOption       |#include <limits.h>
    parseJdwpOptions       |#include <ctype.h>
    enableAssertions       |#include <sys/wait.h>
    dvmLateEnableAssertions|#include <unistd.h>
    freeAssertionCtrl      |
    processXjitop          |#include "Dalvik.h"
    processXjitmethod      |#include "test/Test.h"
    processOptions         |#include "mterp/Mterp.h"
    setCommandLineDefaults |#include "Hash.h"
    busCatcher             |
    blockSignals           |#define kMinHeapStartSize   (1*1024*1024)
    ScopedShutdown [ScopedS|#define kMinHeapSize        (2*1024*1024)
    ~ScopedShutdown [Scoped|#define kMaxHeapSize        (1*1024*1024*1024)
    disarm [ScopedShutdown]|
    dvmStartup             |/*
    registerSystemNatives  | * Register VM-agnostic native methods for system classes.
    initZygote             | */
    dvmInitAfterZygote     |extern int jniRegisterSystemMethods(JNIEnv* env);
    initJdwp               |
    dvmPrepForDexOpt       |/* fwd */
    dvmShutdown            |static bool registerSystemNatives(JNIEnv* pEnv);
    dvmFprintf             |static bool initJdwp();
    dvmPrintNativeBackTrace|static bool initZygote();
    dvmPrintNativeBackTrace|
    dvmAbort               |
                           |/* global state */
~                          |struct DvmGlobals gDvm;
~                          |struct DvmJniGlobals gDvmJni;
 Tag List   3,1        All android4.4/dalvik/vm/Init.cpp                          1,1      Top
:Tlist
```

图 2.6　Vim 外观图

```
Init.cpp (/home/dalvik/andr
  variable
    gDvm
    gDvmJni
    gDvmJit
    gDvmICHitCount
```

图 2.7　tag 中的变量区域

```
  macro
    __STDC_LIMIT_MACROS
    kMinHeapStartSize
    kMinHeapSize
    kMaxHeapSize
```

图 2.8　tag 中的宏区域

```
  class
    ScopedShutdown
```

图 2.9　tag 中的类区域

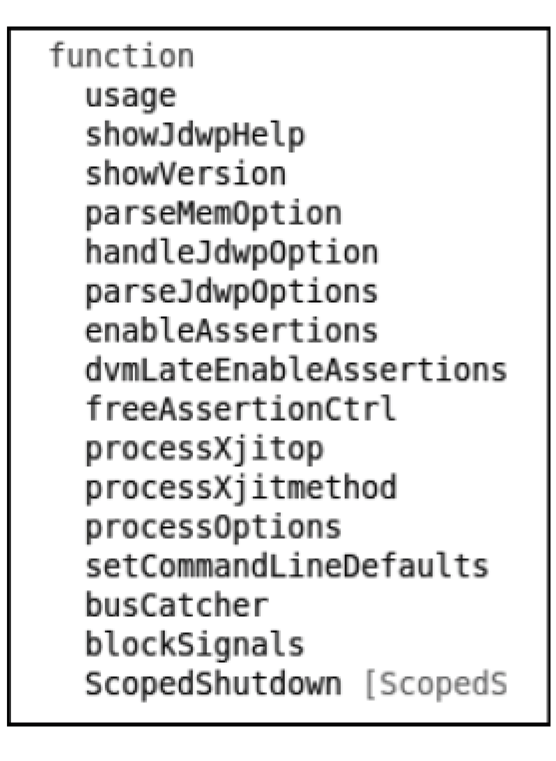


图 2.10　tag 中的函数区域

2.3　Doxygen 工具

Doxygen 能够分析源码中特定格式的注释，并根据这些注释生成源码文档。其优势是：①代码与文档能保持同步；②能对文档做版本管理。同时，Doxygen 静态分析源码，能根据函数调用关系、类间关系等生成 dot 文件。Graphviz 能根据这些 dot 文件生成相应的图。因此在安装时，需要安装 Graphviz。

首先安装 Graphviz 工具。在终端中安装 Graphviz，如图 2.11 所示。

```
dalvik@dalvik-laptop:~/downloads$ sudo apt-get install graphviz
Reading package lists... Done
Building dependency tree
Reading state information... Done
Suggested packages:
  graphviz-doc
The following NEW packages will be installed:
  graphviz
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 435kB of archives.
After this operation, 1,208kB of additional disk space will be used.
Get:1 http://cn.archive.ubuntu.com/ubuntu/ lucid/main graphviz 2.20.2-8ubuntu3 [
435kB]
Fetched 435kB in 0s (541kB/s)
Selecting previously deselected package graphviz.
(Reading database ... 146265 files and directories currently installed.)
Unpacking graphviz (from .../graphviz_2.20.2-8ubuntu3_amd64.deb) ...
Processing triggers for man-db ...
Setting up graphviz (2.20.2-8ubuntu3) ...
```

图 2.11 安装 Graphviz

另外，Doxygen 需要一些依赖，包括 Flex、bison、g++等，如果在第 1 章源码环境搭建时没有安装这些工具，在进行下一步前需要用 apt-get 安装这些工具。安装过程和上述的 Graphviz 是一样的。

接下来安装 Doxygen。到官网下载源码包，目前最新的版本是 1.8.5，并编译安装。步骤如图 2.12 所示。

```
dalvik@dalvik-laptop:~/downloads$ tar zxf doxygen-1.8.5.src.tar.gz
dalvik@dalvik-laptop:~/downloads$ ls
doxygen-1.8.5  doxygen-1.8.5.src.tar.gz  taglist_46.zip
dalvik@dalvik-laptop:~/downloads$ cd doxygen-1.8.5/
dalvik@dalvik-laptop:~/downloads/doxygen-1.8.5$ ./configure && make && sudo make
 install
  Autodetected platform linux-g++...
  Checking for GNU make tool... using /usr/bin/make
  Checking for GNU install tool... using /usr/bin/install
  Checking for dot (part of GraphViz)... using /usr/bin/dot
  Checking for perl... using /usr/bin/perl
  Checking for flex... using /usr/bin/flex
  Checking for bison... using /usr/bin/bison
```

图 2.12 编译安装 Doxygen

编译并安装成功后，应该如图 2.13 所示，表示安装成功。

```
/usr/bin/install -d //usr/local/bin
/usr/bin/install -m 755 bin/doxygen        //usr/local/bin
/usr/bin/install -d //usr/local/man/man1
cat doc/doxygen.1     | sed -e "s/DATE/十一月 2013/g" -e "s/VERSION/1.8.5/g" > do
xygen.1
/usr/bin/install -m 644 doxygen.1 //usr/local/man/man1/doxygen.1
rm doxygen.1
dalvik@dalvik-laptop:~/downloads/doxygen-1.8.5$ 
```

图 2.13 Doxygen 编译安装成功后的提示

为使 Doxygen 能生成文档，需要编写一个配置文件。运行：

```
doxygen -g Doxyfile
```

将在当前目录下生成一个 Doxygen 示例配置文件，名为 Doxyfile。配置文件中对每一个配置参数都做了详细的解释，根据自己需要，修改参数值。主要修改了以下几个方面。

（1）输出语言选择中文。

```
OUTPUT_LANGUAGE=Chinese
```

（2）采用 Java Doc 的注释风格，同时禁用了其他风格，如 QT、C++ 等注释风格。

```
JAVADOC_AUTOBRIEF=YES
```

（3）考虑到 Dalvik VM 中包含几种不同类型的语言，如 Java、C++、Python 和汇编，开启了 OPTIMIZE_OUTPUT_JAVA 和 OPTIMIZE_OIUTPUT_FOR_C。

（4）递归扫描所有文件夹，并分析所有类型的变量和定义。

```
EXTRACT_ALL=YES
```

（5）只输出 HTML 文件。

设置不要生成 Latex 和 Chm。

（6）画出函数之间调用和被调用图、类图、继承关系图等。

```
UML_LOOK=YES
CALL_GRAPH=YES
CALLER_GRAPH=YES
```

（7）因为在 Dalvik VM 中有许多的 bash 脚本，视为 C 语言文件。

```
EXTENSION_MAPPING      =sh=C
```

（8）输入源码文件编码选为 UTF-8。

（9）源码太多，只扫描 h、cpp、java、c 文件。

```
FILE_PATTERNS          =*.c \
                        *.cpp \
                        *.h \
                        *.java
```

（10）在后期发现大部分的函数最后都会归结到一些异常处理函数处，为降低生成的调用图的重复性，去掉了 dvmThrowException、assert、dvmAbort 等异常信息。

```
EXCLUDE_SYMBOLS        =dvmThrowException \
                            assert \
                            dvmAbort
```

（11）在生成函数调用关系图时，需要很大的内存，为减少复杂性，只生成三层的函数调用关系图。

```
MAX_DOT_GRAPH_DEPTH    =3
```

（12）开启预处理宏 WITH_JIT

```
PREDEFINED             =WITH_SELF_VERIFICATION \
                        WITH_JIT
```

同时在配置文件中需要指定输入源码位置和输出文档位置。在配置好后，运行

doxygen，将自动读取该文件夹下的 Doxyfile 并根据配置生成文档。生成的 HTML 文档可以用浏览器直接浏览。

点拨 Doxygen 源码官网地址在 ftp://ftp.stack.nl/pub/users/dimitri/doxygen-1.8.4.src.tar.gz。

文档主要有以下几部分。

（1）文件夹之间的调用关系，如图 2.14 所示。这部分可以比较直观地得到各个文件夹之间的关系以及所属模块。

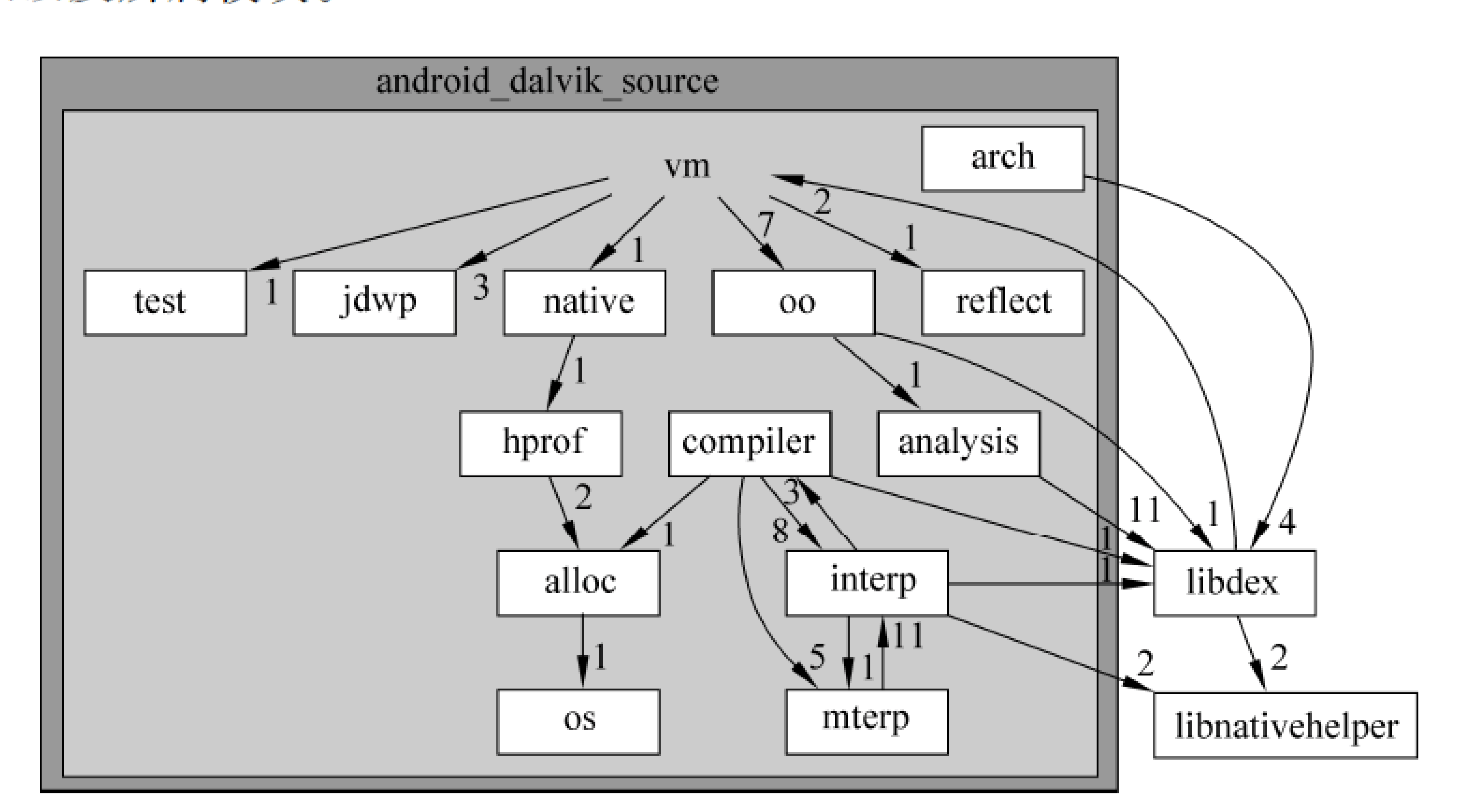


图 2.14 文件夹关系图

（2）类或结构体 UML 图，如图 2.15 所示。

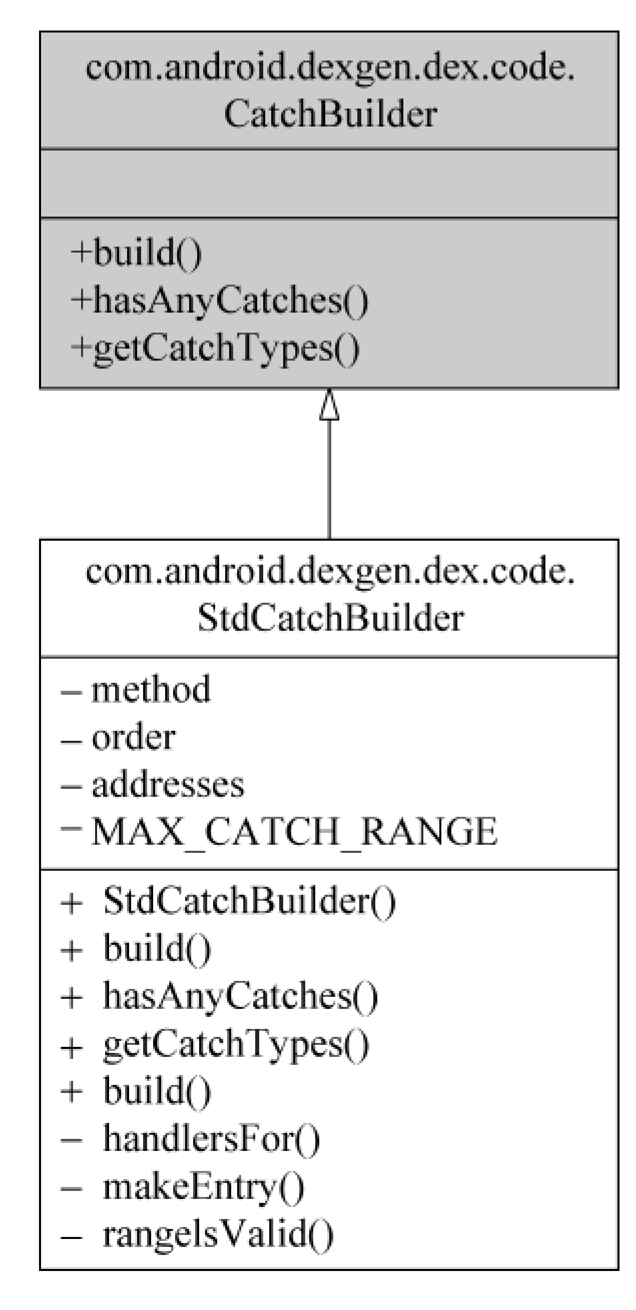


图 2.15 类图和 UML 图

(3) 文件功能和函数功能描述，如图 2.16 和图 2.17 所示。



图 2.16　文件功能展示

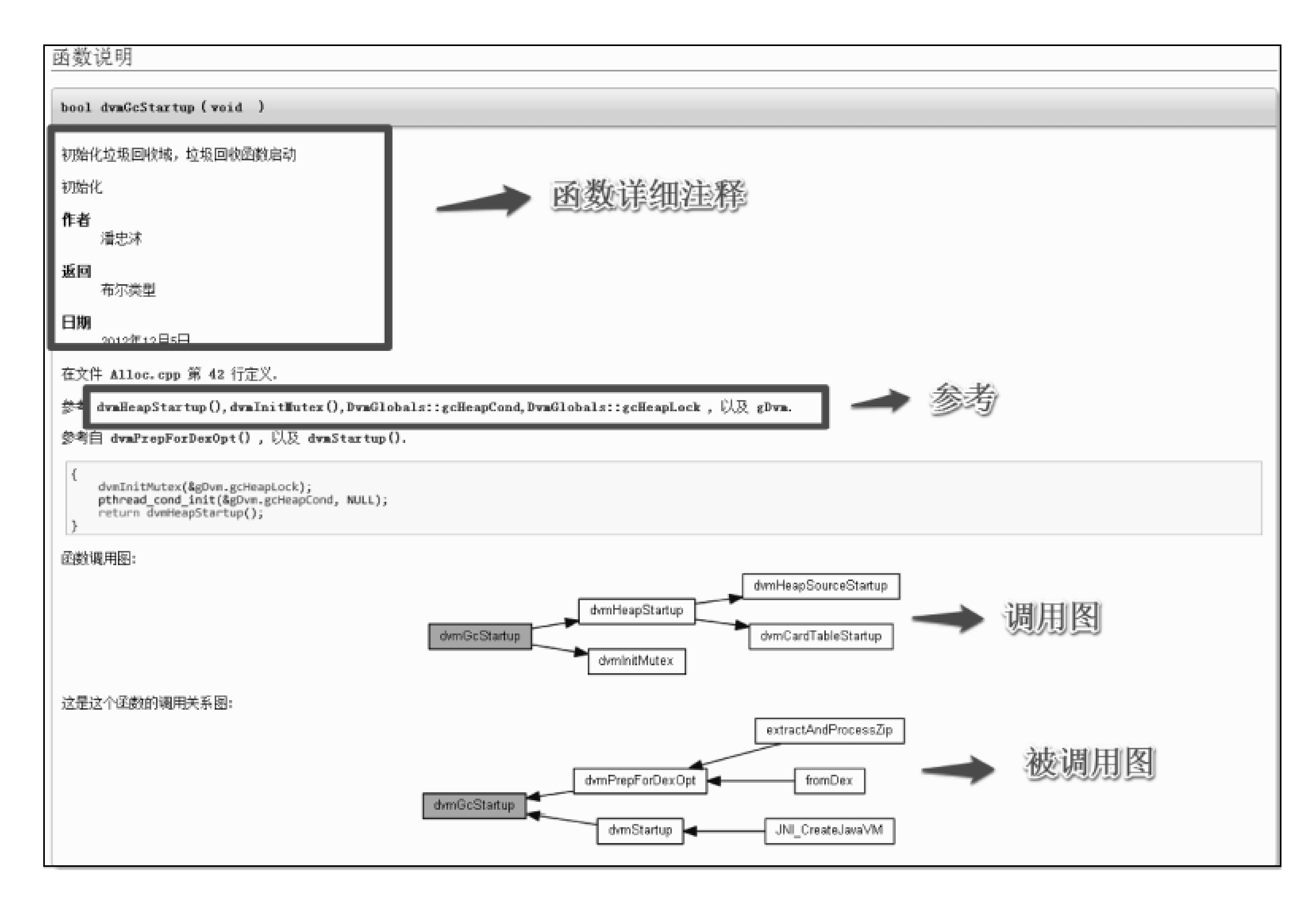


图 2.17　函数功能展示

2.4　GDBSERVER 工具

Android 系统在嵌入式设备上运行，要想对其进行调试，需要配置 Host+Target 环境。Host 端使用 arm-linux-androideabi-gdb，Target 端使用 GDBSERVER。在调试过程中，应

用程序在手机上运行，而 gdb 调试则在 Host 端，也就是 Ubuntu 中。

在 Android 1.0 之后的版本中，GDBSERVER 已经被内置在了源码中。编译完 Android 4.0.4 之后，GDBSERVER 位于

```
~/working/androidsource/android4.0.4/out/target/product/generic/system/bin
```

使用 GDBSERVER 首先需要设置环境变量，根据不同的编译版本设置的环境变量有所不同，但使用的命令是相似的。以 maguro 版本为例，在终端中执行以下命令：

```
. build/envsetup.sh
lunch full-maguro-userdebug
```

如果没有物理机，可以使用模拟器版本，此时 full-maguro-userdebug 变为 full-eng，如图 2.18 所示。

```
dalvik@dalvik-laptop:~/android4.4$ . build/envsetup.sh
including device/moto/stingray/vendorsetup.sh
including device/moto/wingray/vendorsetup.sh
including device/samsung/crespo4g/vendorsetup.sh
including device/samsung/crespo/vendorsetup.sh
including device/samsung/maguro/vendorsetup.sh
including device/samsung/torospr/vendorsetup.sh
including device/samsung/toro/vendorsetup.sh
including device/samsung/tuna/vendorsetup.sh
including device/ti/panda/vendorsetup.sh
including sdk/bash_completion/adb.bash
dalvik@dalvik-laptop:~/android4.4$ lunch

You're building on Linux

Lunch menu... pick a combo:
     1. full-eng
     2. full_x86-eng
     3. vbox_x86-eng
     4. full_stingray-userdebug
     5. full_wingray-userdebug
     6. full_crespo4g-userdebug
     7. full_crespo-userdebug
     8. full_maguro-userdebug
     9. full_torospr-userdebug
     10. full_toro-userdebug
     11. full_tuna-userdebug
     12. full_panda-eng

Which would you like? [full-eng] 8

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.0.4
TARGET_PRODUCT=full_maguro
TARGET_BUILD_VARIANT=userdebug
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a-neon
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=IMM76L
============================================

dalvik@dalvik-laptop:~/android4.4$ █
```

图 2.18 设置环境变量

接下来主要有以下几个步骤。

将测试程序上传到 SD 卡中，如图 2.19 所示。

```
adb push 'HelloWorld.zip' /sdcard/
```

如果遇到问题：

```
failed to copy 'HelloWorld.zip' to '/sdcard//HelloWorld.zip': Read-only file system
```

解决方法：重新挂载文件系统即可。

```
adb shell mount -o remount rw /
```

```
dalvik@dalvik-laptop:~/android4.4$ adb devices
List of devices attached
01498F8E0D005014        device

dalvik@dalvik-laptop:~/android4.4$ adb push 'HelloWorld.zip' /sdcard/
9 KB/s (577 bytes in 0.056s)
dalvik@dalvik-laptop:~/android4.4$ 
```

图 2.19　上传测试程序至 SD 卡中

接着开启 GDBSERVER。运行以下命令以在终端执行需要调试的程序：

```
adb root
adb forward tcp:5039 tcp:5039
adb shell GDBSERVER :5039 dalvikvm -cp /sdcard/HelloWorld.zip HelloWorld
```

终端应该有如下显示，如图 2.20 所示。

```
dalvik@dalvik-laptop:~/android4.4$ adb root
adbd is already running as root
dalvik@dalvik-laptop:~/android4.4$ adb forward tcp:5039 tcp:5039
dalvik@dalvik-laptop:~/android4.4$ adb shell gdbserver :5039 dalvikvm -cp /sdcar
d/HelloWorld.zip HelloWorld
Process dalvikvm created; pid = 745
Listening on port 5039
```

图 2.20　开启 GDBSERVER

然后新建一个终端，在命令行中输入以下代码，在 Host 端打开 gdb，如图 2.21 所示。

```
arm-linux-androideabi-gdb out/target/product/maguro/symbols/system/bin/dalvikvm
```

```
dalvik@dalvik-laptop:~/android4.4$ arm-linux-androideabi-gdb out/target/product/
maguro/symbols/system/bin/dalvikvm
GNU gdb (GDB) 7.1-android-gg2
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-linux-gnu --target=arm-elf-linux".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dalvik/android4.4/out/target/product/maguro/symbols/s
ystem/bin/dalvikvm...done.
(gdb) 
```

图 2.21　开启 gdb

在开启 gdb 后需要设置库路径，分别是：

```
set solib-absolute-prefix out/target/product/maguro/symbos
set solib-search-path out/target/product/maguro/symbos/system/lib
```

之后设置监听端口，需要注意的是，端口号和第二步设置的端口应该是一样的。

```
target remote :5039
```

成功连接上 GDBSERVER 后，两个终端的界面分别如图 2.22 和图 2.23 所示。

```
dalvik@dalvik-laptop:~/android4.4$ adb shell gdbserver :5039 dalvikvm -cp /sdcar
d/HelloWorld.zip HelloWorld
Process dalvikvm created; pid = 745
Listening on port 5039
Remote debugging from host 127.0.0.1
```

图 2.22 成功连接上后手机端的反应

```
dalvik@dalvik-laptop:~/android4.4$ arm-linux-androideabi-gdb out/target/product/
maguro/symbols/system/bin/dalvikvm
GNU gdb (GDB) 7.1-android-gg2
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-linux-gnu --target=arm-elf-linux".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dalvik/android4.4/out/target/product/maguro/symbols/s
ystem/bin/dalvikvm...done.
(gdb)
```

图 2.23 成功连接上 GDBSERVER

接下来就可以设置断点，并跟踪执行了，如图 2.24 所示。

```
(gdb) b main
Breakpoint 1 at 0x8710: file dalvik/dalvikvm/Main.cpp, line 152.
(gdb)
```

图 2.24 设置断点

小结

本章介绍了一些辅助源码分析的工具，包括 Vim、Doxygen、GDBSERVER，并介绍了其使用的方法，为后期的阅读和分析打下了基础。其中，GDBSERVER 的使用一定要熟练掌握，今后大部分的源码分析，都需要这个步骤的验证。

第 3 章

Dex文件及Dalvik字节码格式解析

本章主要内容

✍Dex 文件是什么？

✍Dex 文件具有怎样的特点？

✍Dex 文件具有怎样的结构？

✍Dalvik 字节码和 Java 字节码有怎样的区别且具有怎样的特点？

✍Odex 文件是什么？ 它和 Dex 文件有着怎样的关系？

Dex 文件是 Android 系统的可执行文件，包含应用程序的全部操作指令以及运行时数据。当虚拟机在执行一个应用程序时，需要实时地根据程序需要从 Dex 文件中读取相应的数据以保证程序的正确运行，可以说 Dex 文件是一个 Android 应用的根本。正因为如此，我们需要学习 Dex 文件结构以及内含的 Dalvik 字节码，以更加深入地了解 Dalvik 虚拟机运行机制。

3.1　本章概述

Dex 文件是 Dalvik 虚拟机的可执行文件，由于 Dalvik 是一种针对嵌入式设备而特殊设计的 Java 虚拟机，因此，Dex 文件与标准的 Class 文件在结构设计上有着本质上的区别。另外，Dalvik 运行在资源受限的嵌入式系统中，因此为了适应这种严苛的运行环境，标准的 Java 程序在经过编译后，还需要通过 dx 工具将在编译过程中所生成的数个 Class 文件整合成一个 Dex 文件。这样做的目的就是使其中各个类能够共享数据，在一定程度上降低了冗余，同时也使得文件结构十分紧凑。实验表明，Dex 文件是传统 Jar 文件大小的 50%左右。图 3.1 展示了 Dex 文件结构的特点。

为了进一步提高性能，当一个真实设备在执行目标 Dex 文件之前，需要优化该 Dex 文件并生成与之对应的 Odex 文件，随后该 Odex 文件将替换原 Dex 文件被 Dalvik 虚拟机引用执行。Odex 文件本质上仍然是一个 Dex 文件，只是针对目标平台的特性，在原 Dex 文件的基础上进行了一系列的优化，主要体现在对其内部所包含的字节码进行的一系列处理，主要包括字节码验证，替换优化以及空方法的消除等，使得优化后的 Dex 文件在被虚拟机执行时，能保持相当不错的执行速度。

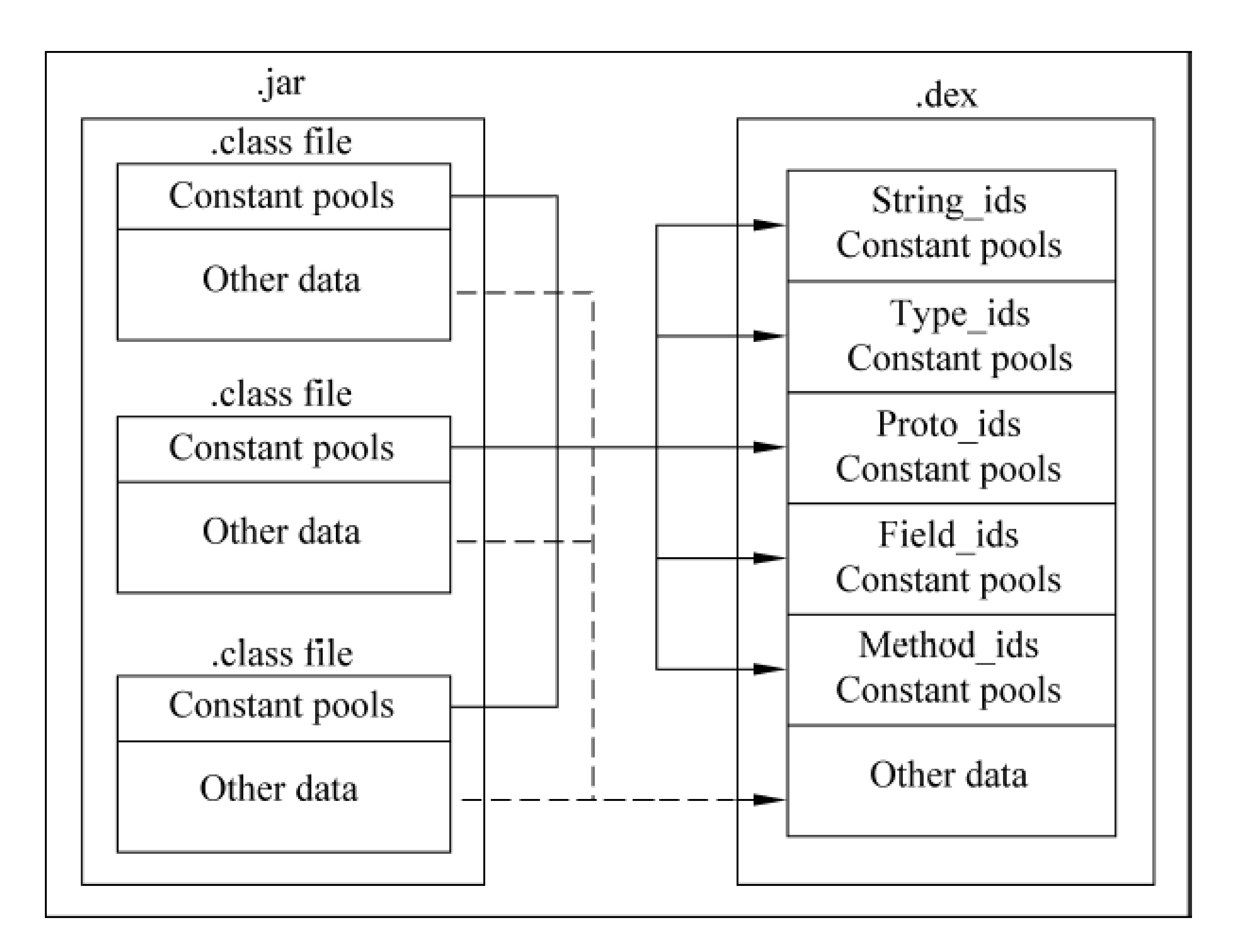


图 3.1 Jar 文件与 Dex 文件结构对比图

3.2 Dex 文件格式

Dex 文件的格式从宏观上讲是非常清晰且明了的，但是如果细究其中各类数据之间的关联关系又是一件非常令人头疼的事情，因此，在本章中在对 Dex 文件进行原理性分析的同时，还将结合一个实际 Dex 文件的反编译结果对这部分内容进行讲解，希望读者可以从理论与实际两方面对 Dex 文件的结构进行理解学习。

点拨 Android 系统为开发者提供了一个 dexdump 工具，其作用是对一个 Dex 文件进行反编译，将其中的二进制数据转化为十六进制数据，并给出相关的辅助注释信息，以帮助使用者更好地理解 Dex 文件结构，该工具的具体使用方法请参见第 4 章。

3.2.1 Dex 文件中的数据结构

在介绍 Dex 文件结构与内容之前，有必要先讲解一下 Dex 文件中所用到的一些基本的数据类型，如表 3.1 所示。

前 8 个属于常用的标准数据结构，在此就不再过多介绍。而 sleb128、uleb128 以及 uleb128p1 是 Dex 文件中特有的数据类型。根据开发文档（位于 Android 源码的 Dalvik/docs 路径下的 dex-format.html）的描述，每一个 leb128 都是由 1～5B 组成，通过这种组合形式表示一个 32 位的数据。对于每一个字节都有如下的规定：最高位为标志位，其余 7 位为有效位，当标志位为 1 时表示需要拼接上第二个字节；当第二个字节的最高位也为 1 时，则需要拼接上第三个字节，以此类推。但需要注意的是，leb128 最多由 5 个字节拼接而成，故第 5 个字节的最高位只能为 0。图 3.2 表示了一个 leb128 数据，它只用到了两个字节，其结构如图 3.2 所示。

由于 Dex 文件结构的内容和 leb128 原理实现关系不大，故不再对 leb128 数据类型的源码实现进行过多的介绍，其源码位于 dalvik/libdex/Leb128.h 文件中，有兴趣的读者可以对该数据结构的实现源码进行研究学习，已明确其中的技术细节。

表 3.1　Dex 文件所应用的基础数据结构

类　　型	含　　义
byte	占用 8b,表示 1B 的有符号数
ubyte	占用 8b,表示 1B 的无符号数
short	占用 16b,表示 2B 的有符号数,低字节序
ushort	占用 16b,表示 2B 的无符号数,低字节序
int	占用 32b,表示 4B 的有符号数,低字节序
uint	占用 32b,表示 4B 的无符号数,低字节序
long	占用 64b,表示 8B 的有符号数,低字节序
ulong	占用 64b,表示 8B 的无符号数,低字节序
sleb128	有符号 LEB128,其长度可变,1～5B
uleb128	无符号 LEB128,其长度可变,1～5B
uleb128p1	无符号 LEB128 值加 1,其长度可变,1～5B

Bitwise diagram of a two-byte LEB128 value															
First byte								Second byte							
1	bit_6	bit_5	bit_4	bit_3	bit_2	bit_1	bit_0	0	bit_{13}	bit_{12}	bit_{11}	bit_{10}	bit_9	bit_8	bit_7

图 3.2　leb128 数据结构示意图

点拨　在实际的源码编写中,开发人员常用 uX 表示数据结构中成员变量的数据类型,其中 X 的值可以为 1、2、4,分别表示 1 字节数、2 字节数以及 4 字节数。

3.2.2　Dex 文件结构分析

从宏观上讲,Dex 文件的结构很简单,实际上就是由多个不同的结构体数据以首尾相接的方式拼凑而成。首先直观地介绍一下 Dex 文件都包含哪些数据、数据是以怎样的方式排列以及这些数据的主要功能,如表 3.2 所示。

表 3.2　Dex 文件各部分数据含义

数据名称	含　　义
header	Dex 文件的头部,记录 Dex 文件的相关属性
string_ids	字符串数据索引,记录了各个字符处在数据区的地址偏移量
type_ids	类型数据索引,记录了各个类型的字符串索引
proto_ids	原型数据索引,记录了方法声明的字符串、返回类型字符串、参数列表
field_ids	字段数据索引,记录了所属类、声明类型以及方法名等信息
method_ids	类方法索引,记录方法所属类名、方法声明以及该方法名等信息
class_defs	类定义数据,记录了指定类各类信息,包括接口、超类、类数据偏移量等
data	数据区,保存着各个类的真实数据
link_data	连接数据区

根据表 3.2 可以对 Dex 文件的整体结构有一个直观的了解，但是仍不能明确 Dex 文件内部数据之间的关联关系，下面将逐一对以上各个部分数据进行介绍。

1. header

header 是 Dex 文件的文件头，简单地记录了 Dex 文件的一些基本信息，以及其大致的数据分布。Dex 文件头部的总长度是固定的 0x70，其中每一信息项所占用的内存空间也是相应固定的，例如，“magic”字段所占用的内存空间即为 8B。这样做的好处是，虚拟机在处理目标 Dex 文件的初期，可以不用考虑 Dex 文件的多样性，而根据约定俗成的规则读取文件头，即可获取目标 Dex 文件的一个大致信息。表 3.3 展示了 Dex 文件头的内部结构以及所包含的各类数据。

表 3.3 Dex 文件的头部的信息

字段名称	偏移值	长度	描述
magic	0x0	8	“Magic”值，即魔数字段，格式如“dex/n035/0”
Cheaksum	0x8	4	校验码
signature	0xC	20	SHA-1 签名
File_size	0x20	4	Dex 文件总长度
Header_size	0x24	4	文件头长度，009 版本＝0x5C，035 版本＝0x70
Endian_tag	0x28	4	标示字节顺序的常量
Link_size	0x2C	4	链接段的大小，如果为 0 就是表示是静态链接
Link_off	0x30	4	链接段的开始位置
Map_off	0x34	4	Map 数据基址
String_ids_size	0x38	4	字符串列表中字符串个数
String_ids_off	0x3C	4	字符串列表基址
Type_ids_size	0x40	4	类列表里类型个数
Type_ids_off	0x44	4	类列表基址
Proto_ids_size	0x48	4	原型列表里原型个数
Proto_ids_off	0x4C	4	原型列表基址
Field_ids_size	0x50	4	字段列表里字段个数
Field_ids_off	0x54	4	字段列表基址
Method_ids_size	0x58	4	方法列表里方法个数
Method_ids_off	0x5C	4	方法列表基址
Class_defs_size	0x60	4	类定义表中类的个数
Class_defs_off	0x64	4	类定义列表基址
Data_size	0x68	4	数据段的大小，必须以 4 字节对齐
Data_off	0x6C	4	数据段基址

2. string_ids

这一区域中存储的是 Dex 文件字符串资源的索引信息，实际上，该索引信息是目标字符串在 Dex 文件数据区所在的真实物理偏移量。读者可能会问，虚拟机是通过什么方式读取这个索引信息的呢？还记得在介绍 header 的时候，曾提到虚拟机需要一种约定俗成的规则，才能对目标数据进行正确的访问，这个思想在这部分数据区也是成立的。在本区域中，Google 的开发人员规定了一种结构体——DexStringId，先看下其函数定义：

代码清单 3.1　dalvik/vm/libdex/DexFile.h:DexStringId 函数定义

```
Struct DexStringId{
   u4 stringDataOff;                /* 在 Dex 文件中的实际偏移量 * /
};
```

该数据结构只有 stringDataOff 一个成员变量，其记录了目标字符串在 Dex 文件中的实际偏移量，当虚拟机想要读取该字符串时，只需将 Dex 文件在内存中的起始地址加上 stringDataOff 所指的偏移量，即可得到该字符串在内存中的实际物理地址。代码清单 3.2 是一段 Dex 文件中经过反编译的 string_ids 字符串索引信息。

点拨　在 Dex 文件中，每个字符串都对应一个 DexStringId 数据结构，该数据结构的大小为 4B，是一个确定的量。同时，虚拟机可以通过头文件中的 String_ids_size 变量知道当前 Dex 文件中字符串的总数，也就是 string_ids 区域中 DexStringId 数据结构的总数，因此，虚拟机通过简单的乘法运算即可实现对该索引资源进行正确的访问。

下面是一段 Dex 文件中经过反编译的 string_ids 字符串索引信息。

代码清单 3.2　string_ids 字符串索引资源实例

```
000070:    4202 0000    |   string_data_off: 00000242   [0] " j:"
000074:    4702 0000    |   string_data_off: 00000247   [1] "<init>"
000078:    4f02 0000    |   string_data_off: 0000024f   [2] "I"
00007c:    5202 0000    |   string_data_off: 00000252   [3] "L"
000080:    5502 0000    |   string_data_off: 00000255   [4] "LI"
000084:    5902 0000    |   string_data_off: 00000259   [5] "LL"
000088:    5d02 0000    |   string_data_off: 0000025d   [6] "LLoop2;"
……
0000b8:    0803 0000    |   string_data_off: 00000308   [12] "main"
0000bc:    0e03 0000    |   string_data_off: 0000030e   [13] "out"
0000c0:    1303 0000    |   string_data_off: 00000313   [14] "println"
0000c4:    1c03 0000    |   string_data_off: 0000031c   [15] "toString"
```

其中，左边两列信息分别表示的是 Dex 文件内部地址以及存储的实际内容，而右边是 dexdump 工具所给出的辅助注释信息。下面依据这段反编译数据对 string_ids 字符串索引信息区域进行介绍。

首先观察左边两列数据，在前面曾介绍 Dex 文件头 header 的长度即为固定的 0x70，所以 Dex 文件中字符串索引区第一项数据的起始地址即为 000070，该项数据内容是 4202 0000，根据高位在后的存储规则，经过整理后这段数据即为 00000242，我们发现该段数据占

用了 4B,正好对应着前面所分析的 DexStringId 数据结构的大小。而数据 00000242 代表的是什么意思呢?我们看到右边的辅助注释信息 string_data_off: 00000242 [0] " j:",其中 string_data_off: 00000242 说明该 00000242 数据表示的是字符串数据在 Dex 文件中的地址偏移量;而" j:"表示的是正是该字符串;[0]表示的是该字符串序号。

为了证明分析的正确性,直接定位到 Dex 文件的 00000242 地址处,其十六进制数据如下:

代码清单 3.3 字符串数据的实际存储实例

```
000242: 03                  |utf16_size: 00000003
000243: 206a 3a00           |" j:"
```

该读取过程如下:虚拟机首先根据字符串索引信息获得第一条字符串的 Dex 文件内部地址偏移,即 00000242,随后虚拟机将读取 00000242 位置的第一个字节,在本例中该值为 03,即表示从 00000242 后面一个地址 00000243 开始至 00000246 为该字符串的实际存储位置,其值 206a 3a00 经过转化即可得到字符串“j:”,虚拟机通过这种方式即可获取到一个字符串。

3. type_ids

在这一区域中存储的是类型资源的索引信息,本着和 string_ids 区域中相同的设计思想,开发人员也为其规定了相应的数据结构——DexTypeId 用于存储这部分信息。首先观察一下该数据结构的函数定义。

代码清单 3.4 dalvik/vm/libdex/DexFile.h:DexTypeId 数据结构函数定义

```
Struct DexTypeId{
   u4 descriptorIdx;            /* 指向字符串索引表 */
};
```

在 Dex 文件中,类型是以字符串的形式保存在数据区中,因此,DexTypeId 数据结构中的 descriptorIdx 变量保存的是目标类型在字符串索引表中的序列号。

在头文件中,Type_ids_size 变量用于标示 type_ids 资源区中所记录的类型总数量,即 DexTypeId 数据结构的个数,和 DexStringId 一样,该数据结构的大小也是固定为 4B。因此,当数量的单位与大小都明确的前提下,虚拟机也可以实现对这部分资源的正确访问。

在这里,同样根据一段实例数据讲解类型索引资源的实际构造与应用。

代码清单 3.5 type_ids 类型索引资源实例

```
000c8: 0200 0000      |  descriptor_idx: 00000002 | [0]I
0000cc: 0600 0000     |  descriptor_idx: 00000006 | [1]LLoop2
0000d0: 0700 0000     |  descriptor_idx: 00000007 | [2]Ljava/io/PrintStream
0000d4: 0800 0000     |  descriptor_idx: 00000008 | [3]Ljava/lang/Object
0000d8: 0900 0000     |  descriptor_idx: 00000009 | [4]Ljava/lang/String
0000dc: 0a00 0000     |  descriptor_idx: 0000000a | [5]Ljava/lang/StringBuilder
0000e0: 0b00 0000     |  descriptor_idx: 0000000b | [6]Ljava/lang/System
0000e4: 0d00 0000     |  descriptor_idx: 0000000d | [7]V
0000e8: 0f00 0000     |  descriptor_idx: 0000000f | [8][Ljava/lang/String
```

以上为类型索引资源的反编译数据，左边一列为 Dex 文件内部偏移地址，中间和右边一列为反编译生成的辅助信息。该部分资源和字符串索引资源的存储方式相似，都是以相应的数据结构为单位顺次存储。以该表的第一项为例，虚拟机首先读取 0000c8 地址的数据，该数据即为第一个 DexTypeId 数据结构中 descriptorIdx 成员变量所保存的数据，在本例中该值为 0200 0000，根据数据高位在后的存储原则，该值实际为 00000002，即表示为字符串索引表中的 00000002 项，可以通过前面介绍的方法，找出用于表示该类型的字符串，具体的过程就不再赘述。

经过实际查找发现，用于表示该类型的字符串确实为“I”，即和反编译生成的辅助信息相同。读者可以根据手边的实例 Dex 文件对存储结构进行验证。

4. proto_ids

在这一区域中存储的内容是原型资源的索引信息，数据结构 DexProtoId 将负责规格化这些索引信息，DexProtoId 数据结构的函数定义如下。

代码清单 3.6　dalvik/vm/libdex/DexFile.h:DexProtoId 数据结构函数定义

```
Struct DexProtoId{
   u4 ShortyIdx;              /* 方法声明字符串,指向字符串索引表 */
   u4 returnTypeIdx;          /* 方法返回类型,指向字符串索引表 */
   u4 parameterOff;           /* 参数列表,该列表是一个 DexTypeList 数据结构 */
};
```

相比于前面两种信息资源的存储格式，原型数据索引信息相对较为复杂，前两个变量较好理解，其所代表的真实资源都是字符串。parameterOff 变量就相对复杂，其指向一个 DexTypeList 数据结构，而该数据结构的函数定义如下。

代码清单 3.7　dalvik/vm/libdex/DexFile.h:DexTypeList 数据结构函数定义

```
Struct DexTypeList{
   u4 size;                   /* 表示 DexTypeItem 数据结构的个数 */
   DexTypeItem list[size];    /* DexTypeItem 数据结构列表 */
   };
```

可以看到，其中 size 变量记录了方法的参数的个数，下面的 DexTypeItem 数据结构列表则依次记录了各个参数的类型。再看下 DexTypeItem 数据结构的函数定义。

代码清单 3.8　dalvik/vm/libdex/DexFile.h:DexTypeItem 数据结构函数定义

```
Struct DexTypeItem{
   u2 typeIdx;                /* 表示参数的类型,其指向 DexTypeId 索引表 */
    };
```

其中，typeIdx 变量指向类型资源索引表，虚拟机通过该变量即可获取相应参数的类型。

看到这里，想必有读者会有一个疑问，DexTypeList 数据结构中所记录的 DexTypeItem 数量是不确定的，因此会导致 DexTypeList 数据结构的对象的大小也不能确定，最终是不是也会使 DexProtoId 数据结构的实例对象也会出现大小不一的情况，如果是，那么虚拟机该如何访问不规则的 proto_ids 数据资源？

Google 的开发者当然不会让这种情况出现，回过头仔细观察一下 DexProtoId 数据结构的第三个成员变量 parameterOff，其数据类型是 u4，表示占用 4B 的内存资源，应该是一个 Dex 文件内部的偏移地址，该地址就应该是其 DexTypeList 数据结构的偏移地址。因此，对于 DexProtoId 数据结构来说，其大小也是一个固定值，即为 12B。从头文件的 Proto_ids_size 变量中获取原型方法的数量，也就是接下来 DexProtoId 数据结构的个数，故虚拟机也可以根据一贯的原则实现对 proto_ids 区域的数据进行正确的访问。

相对前面两类资源，原型索引资源相对复杂，下面仍通过一段实例数据进行讲解。

代码清单 3.9　Dex 文件中 proto_ids 类型索引资源实例

```
|[0] java.lang.String proto()
0000ec: 0300 0000   |   shorty_idx:         00000003 // "L"
0000f0: 0400 0000   |   return_type_idx:    00000004 // java.lang.String
0000f4: 0000 0000   |   parameters_off:     00000000
|[1] java.lang.StringBuilder proto(int)
0000f8: 0400 0000   |   shorty_idx:         00000004 // "LI"
0000fc: 0500 0000   |   return_type_idx:    00000005 // java.lang.StringBuilder
000100: 2c02 0000   |   parameters_off:     0000022c
|[2] java.lang.StringBuilder proto(java.lang.String)
000104: 0500 0000   |   shorty_idx:         00000005 // "LL"
000108: 0500 0000   |   return_type_idx:    00000005 // java.lang.StringBuilder
00010c: 3402 0000   |   parameters_off:     00000234
```

前面讲到，DexProtoId 数据结构的大小为 12B，以第一表项 java. lang. String 的原型索引资源为例，其数据地址起始于 0000ec，结束于 0000f7，数据总量正好为 12B。在这部分数据中，前 4 个字节的数据为 DexProtoId 数据结构成员变量 ShortyIdx 所存储的值，该数据表示字符串索引表中表项的序号，中间 4 个字节数据为成员变量 returnTypeIdx 所存储的值，和 ShortyIdx 变量一样表示字符串索引表中表项的序号以及后面 4 个字节数据为成员变量 parameterOff 所存储的值，该值是一个 Dex 文件的内部地址，该地址直接指向前面讲到的 DexTypeList 数据结构。由于分析方法相似，故本书就不再对 DexTypeList 数据结构以及和它相关的 DexTypeItem 数据结构进行实例分析，但仍希望读者可以利用空闲时间，通过一个经过反编译的 Dex 文件对上面两种未介绍的数据结构进行研究，以达到学练结合的目的。

通过上面的分析，发现在原型资源索引区中，其数据存储方式也是以结构体为单位进行顺序排列。

点拨　事实上，在 Dex 文件中的数据都是以数据结构顺序排列的形式保存的，虽然不同部分的数据之间没有明确的界限，但虚拟机可以通过 Dex 文件头获取相关数据结构的数量，同时各部分资源的数据结构在内存中所占用的宽度是固定的，因此虚拟机只需要通过简单的乘法运算即可获得各部分资源的实际存储位置，由此便可以实现对各部分资源的正确访问。

5. field_ids

在这一区域中存储着字段资源的索引信息，在这里用到数据结构 DexFieldId 实现对字

段资源索引信息的规格化，先观察一下该数据结构的函数定义。

代码清单 3.10　dalvik/vm/libdex/DexFile.h:DexFieldId 数据结构函数定义

```
Struct DexFieldId{
    u2 classIdx;            /* 标示所属类的类型 */
    u2 typeIdx;             /* 标示该字段类型 */
    u4 nameIdx;             /* 字段名称 */
};
```

classIdx 变量记录了字段所属类的类型，其指向类型索引表 DexTypeId 一个表项；变量 typeIdx 记录了该字段的类型，其也是指向类型索引表 DexTypeId 一个表项；nameIdx 变量记录了该字段的名称，故其指向字符串索引表 DexStringId 的一个表项。

DexFieldId 数据结构的大小为 8B，虚拟机可以通过头文件中 Field_ids_size 变量获取 Dex 文件中字段个数，即在 field_ids 区域中 DexFieldId 结构体的个数。与其他区域的资源一样，都是采取对多个数据结构进行排列的方式，保持了高度的规格化。

下面是一段 Dex 文件中字段索引信息的实例数据。

代码清单 3.11　Dex 文件中 field_ids 字段索引资源实例

```
|[0] java.lang.System.out:Ljava/io/PrintStream;
000134: 0600        |   class_idx: 0006
000136: 0200        |   type_idx:  0002
000138: 1300 0000   |   name_idx:  00000013
```

从上面数据的左半部分，可以清楚地看到：在这部分数据中，前两个字节的数据为 DexFieldId 结构体成员变量 classIdx 所保存的内容，即为类型索引表中某一表项的序号，往后两个字节的数据为成员变量 typeIdx 所保存的内容，即类型索引表中某一表项的序号，最后 4 个字节则为成员变量 nameIdx 所保存的内容，即为字符串索引表中某一表项的序号。

6. method_ids

method_ids 资源区保存了 Dex 文件中类方法数据的索引信息，其采用数据结构 DexMethodId 对这部分索引信息进行规格化，其具体的函数定义如下。

代码清单 3.12　dalvik/vm/libdex/DexFile.h:DexMethodId 数据结构函数定义

```
Struct DexMethodId{
    u2 classIdx;            /* 标示所属类的类型 */
    u2 protoIdx;            /* 标示方法原型类型 */
    u4 nameIdx;             /* 方法名称 */
};
```

classIdx 变量记录了该方法所属类的类型，其指向类型索引表 DexTypeId 一个表项；变量 protoIdx 记录了该方法的原型，其是指向原型索引表 DexProtoId 一个表项；nameIdx 变量记录了该字段的名称，故其指向字符串索引表 DexStringId 的一个表项。

和 DexFieldId 数据结构一样，DexMethodId 的大小也是 8B，虚拟机可以通过头文件中的 Method_ids_size 变量获取 Dex 文件方法数量，即方法资源索引区中 DexMethodId 数据

结构的个数。

同样，这部分资源也是规格化的，虚拟机可以对其正确读取。下面从一段真实 Dex 文件数据的角度，再次对这部分内容进行介绍，下面是一段类方法索引信息数据。

代码清单 3.13 Dex 文件中 Method_ids 方法索引资源实例

```
|[0] Loop2.<init>:()V
00013c: 0100            |  class_idx: 0001
00013e: 0300            |  proto_idx: 0003
000140: 0100 0000       |  name_idx:  00000001
|[1] Loop2.main:([Ljava/lang/String;)V
000144: 0100            |  class_idx: 0001
000146: 0500            |  proto_idx: 0005
000148: 1200 0000       |  name_idx:  00000012
|[2] java.io.PrintStream.println:(Ljava/lang/String;)V
00014c: 0200            |  class_idx: 0002
00014e: 0400            |  proto_idx: 0004
000150: 1400 0000       |  name_idx:  00000014
```

以第一个类方法 Loop2.<init>为例，我们知道 DexMethodId 数据结构的大小为 8B，通过观察上面的数据段，对于第一个类方法 Loop2.<init>，其类方法索引信息的数据段起始于 00013c 并且终止于 000143，正好占据着 8B 的内存空间。因此，可以确定 00013c 和 00013d 两个地址中的数据对应着 classIdx 变量所保存的值，即指向类型索引表 DexTypeId 一个表项，00013e 和 00013f 两个地址中的数据对应着 protoIdx 变量中所保存的值，即指向原型索引表 DexProtoId 一个表项以及从 000140 开始以后 4B 中所保存的内容对应着变量 nameIdx，该值指向字符串索引表 DexStringId 中的某一表项。

7. class_defs

在 class_defs 资源区中，使用数据结构 DexClassDef 对资源进行规格化，先看一下其具体的函数定义。

代码清单 3.14 dalvik/vm/libdex/DexFile.h:DexClassDef 数据结构函数定义

```
Struct DexClassDef{
    u4 classIdx;            /* 标示所属类的类型,指向类型索引表 */
    u4 accessFlags;         /* 访问标示符 */
    u4 superclassIdx;       /* 超类类型,指向类型索引表 */
    u4 interfaceOff;        /* 接口信息,指向一个 DexTypeList 数据结构实例 */
    u4 sourceFiledx;        /* 表示源文件名,指向字符串索引表 */
    u4 classDataOff;        /* 类数据,指向一个 DexClassData 数据结构实例 */
    u4 staticValuesOff;     /* 静态值偏移量 */
};
```

从上面的 DexClassDef 数据结构的函数定义中可以发现，该 class_defs 资源明显较之前几类资源在结构上要复杂很多，在一定程度上来说，该数据结构似乎涵盖了一个类所需的全部资源。在这 7 个成员变量中，classDataOff 变量是最重要的核心内容，该变量所记录的

是一个 Dex 文件内部地址的偏移量，该地址指向一个 DexClassData 数据结构实例。先观察一下其函数定义。

代码清单 3.15 dalvik/vm/libdex/DexClass.h：DexClassData 数据结构函数定义

```
Struct DexClassData{
    DexClassDataHeader  header;          /* 类数据头 */
    DexField*   staticFields;            /* 指向目标的静态字段 */
    DexField*   instanceFields;          /* 指向目标类的实例字段 */
    DexMethod*   directMethod;           /* 指向目标类直接方法 */
    DexMethod*   virtualMethod           /* 指向目标类直接方法 */
};
```

该数据结构的功能正如其名称一样，用于记录目标类在 Dex 文件中目标类数据，其中 header 变量用于记录目标类各类数据的概况，其函数定义如下。

代码清单 3.16 dalvik/vm/libdex/DexClass.h：DexClassDataHeader 数据结构函数定义

```
Struct DexClassDataHeader{
u4 staticFieldsSize;                     /* 记录静态字段的个数 */
u4 intanceFieldsSize;                    /* 记录实例字段的个数 */
u4 directMethodSize;                     /* 记录直接方法的个数 */
u4 virtualMethodSize;                    /* 记录虚方法的个数 */
};
```

该头部信息记录了目标类中各个部分数据的个数，主要包括：静态字段、实例字段、直接方法以及虚方法。其中，staticFieldsSize 变量值加上 intanceFieldsSize 变量值即为接下来 DexField 数据结构的数量；directMethodSize 变量值加上 virtualMethodSize 变量值即为接下来 DexMethod 数据结构的数量。

DexField 数据结构的函数定义如下。

代码清单 3.17 dalvik/vm/libdex/DexClass.h：DexField 数据结构函数定义

```
Struct DexField{
    u4 fieldIdx;                 /* 指向字段索引表中的一个表项，即一个 DexFieldId 数据结构 */
    u4 accessFlags;              /* 访问标示符 */
};
```

读到这里，想必很多读者都有一个疑问，到目前所介绍的数据结构似乎都是用于对类数据的索引，那么真实的 Dalvik 字节码究竟在哪儿？接下来就介绍一个重量级成员 DexMethod 数据结构，首先看下它的函数定义。

代码清单 3.18 dalvik/vm/libdex/DexClass.h：DexMethod 数据结构函数定义

```
Struct DexMethod{
    u4 methodIdx;                /* 指向方法索引表中的一个表项，一个 DexMethodId 数据结构 */
    u4 accessFlags;              /* 访问标示符 */
    u4 codeOff;                  /* 指向一个 DexCode 数据结构 */
};
```

通过 DexMethod 数据结构，让我们离真实的字节码数据又进了一步，前两个变量较好理解，而 codeOff 变量是什么含义呢？其实，codeOff 正是指向了一个 DexCode 数据结构，而该数据结构正是用来记录 Dex 文件中目标类方法字节码，下面让我们一起来揭开它的面纱。DexCode 数据结构函数定义如下。

代码清单 3.19 dalvik/vm/libdex/DexFile.h：DexCode 数据结构函数定义

```
Struct DexCode{
  u2 registerSize;               /* 寄存器个数 */
  u2 insSize;                    /* 输入参数个数 */
  u2 outsSize;                   /* 外部方法使用寄存器数 */
  u2 triesSize                   /* tries 个数 */
  u4 debugInfoOff                /* 调试信息地址 */
  u4 insnsSize                   /* 方法指令个数 */
  u2 insns[insnsSize]            /* 真实指令数组 */
};
```

Dex 文件通过 DexCode 数据结构管理类方法的全部执行信息，其中 registerSize 记录了方法执行期间所需的寄存器总数，insSize 记录了方法的入口参数的个数，outsSize 记录了方法使用寄存器数，triesSize 记录了 tries 个数，debugInfoOff 记录了方法的调试信息，insnsSize 记录了方法字节码数量，即接下来保存的指令个数。

class_defs 数据和前面几类资源在 Dex 文件中的存储方式相同，都是以结构体为单位顺序存储，DexClassDef 数据结构数据量需占用 28B 的内存空间，其中每个成员变量占用 4B 的内存空间。

下面是一段 Dex 文件中实际的 ClassDef 资源数据段。

代码清单 3.20 Dex 文件中 class_defs 数据资源实例

```
  |[0] Loop2
00017c: 0100 0000    |  class_idx:          00000001
000180: 0100 0000    |  access_flags:       public
000184: 0300 0000    |  superclass_idx:     00000003 // java.lang.Object
000188: 0000 0000    |  interfaces_off:     00000000
00018c: 0c00 0000    |  source_file_idx:    0000000c // Loop2.java
000190: 0000 0000    |  annotations_off:    00000000
000194: 3803 0000    |  class_data_off:     00000338
000198: 0000 0000    |  static_values_off:  00000000
```

对上面数据段的左半部分进行简单的分析即可得出结论，在实际中 DexClassDef 数据结构也确实是以源码定义的规则进行存储的，有兴趣的读者可以结合自己编写的 Dex 文件对该数据结构的存储方式进行验证。

前面介绍 DexClassDef 数据结构时，曾提到变量 classDataOff 非常重要，在前面的实际数据段中，该变量所保存的数值为 00000338，因此我们直接定位到该实例 Dex 文件的 00000338 字节处，得到如下数据段。

代码清单 3.21 Dex 文件中 class data 数据资源实例

```
|[338] class data for Loop2
000338: 00                  |  static_fields_size:   00000000
000339: 00                  |  instance_fields_size: 00000000
00033a: 02                  |  direct_methods_size:  00000002
00033b: 00                  |  virtual_methods_size: 00000000
```

首先看到地址 000338～00033b 分别存储了目标类中的静态字段、实例字段、直接方法以及虚方法的数量。在本例中，静态方法的数量为 2，意味着 Loop2 类中包含两个直接方法，下面的数据段为这两个直接方法对应的 DexMethod 数据结构实例。

代码清单 3.22　Dex 文件中 class method 数据资源实例

```
                            |  direct_methods:
                            |  [0] Loop2.<init>:()V
00033c: 00                  |    method_idx:    00000000
00033d: 8180 04             |    access_flags: public|constructor
000340: 9c03                |    code_off:      0000019c
                            |  [1] Loop2.main:([Ljava/lang/String;)V
000342: 01                  |    method_idx:    00000001
000343: 09                  |    access_flags: public|static
000344: b403                |    code_off:      000001b4
```

以第一个方法即 Loop2 类的构造方法的索引信息为例，其中第一个字段为 0，表示在 DexMethodId 表中的第一项，即 Loop2 的<init>方法；第二个字段为 8180 04，表示该方法为一个 public 类型的构造函数；第三个字段为 9c03，经过转化，该字段实际存储的值为 0000019c，该值为 Dex 文件的内部偏移地址，其直接指向用于描述该方法的 DexCode 数据结构实例，我们定位到地址 0000019c 处，其数据段如下。

代码清单 3.23　Dex 文件中 Dalvik 操作码数据资源实例

```
                            |[19c] Loop2.<init>:()V
00019c: 0100                |  registers_size:  0001
00019e: 0100                |  ins_size:        0001
0001a0: 0100                |  outs_size:       0001
0001a2: 0000                |  tries_size:      0000
0001a4: 2603 0000           |  debug_off:       00000326
0001a8: 0400 0000           |  insns_size:      00000004
0001ac: 7010 0300 0000      |  0000: invoke-direct{v0}, java.lang.Object.<init>:()V
0001b2: 0e00                |  0003: return-void
                            |  0004: code-address
                            |    debug info
                            |    line_start: 1
                            |    parameters_size: 0000
                            |    0000: prologue end
                            |    0000: line 1
                            |    end sequence
```

在上面的例子中，可以看到从地址 00019c 至 0001a7 中保存着目标方法在执行过程中

所使用的寄存器个数、输入参数个数、tries 个数以及方法调试信息地址等。紧接着从地址 0001a8 开始，Dex 文件花费 4B 的存储空间保存了目标方法的指令个数，在本例中 insns_size 为 4，故在接下来的存储空间中将花费 8B 存储这 4 条方法指令（每条指令占用 2B 的内存空间）。

在 Dex 文件中，这 4 条指令为 7010 0300 0000 0e00，根据上面右侧的反编译辅助数据，可以清楚前 6 个字节的数据所对应的指令为 invoke-direct {v0}, java. lang. Object. <init>:()V，该指令功能就是对一个类对象实例进行初始化，后面两个字节的数据所对应的指令为 return-void，该指令表示函数返回。

至此，完成了对 Dex 文件中的一系列的数据结构的介绍，并在一个实际的 Dex 文件中根据所总结的理论知识，成功地追踪到了文件中某一个类中某一个方法指令的实际存储位置，并找到了用于实现该方法的 Dalvik 指令。

相信通过上面的介绍，读者应该对 Dex 文件的结构不再陌生，而且在一定程度上了解了 Dex 文件中关键的几类数据存储格式以及各部分数据的关联关系。但作者仍然希望并建议读者在学习完本部分知识后，可以利用空闲时间选取一个实例 Dex 文件，并结合本书所讲解的相关知识对该 Dex 文件进行一次深入的解析，这样可以在实际动手的过程中加深对 Dex 文件结构的理解，升华所学的知识。

3.3 Dalvik 字节码介绍

Dalvik 字节码是专为 Dalvik VM 设计的一种指令集，包含在二进制文件中（Dex 文件）。其演化自 Java 字节码（存储于 class 文件），是一种"跨平台"的中间代码，可以在支持 Dalvik VM 的多重平台上运行。和 Java 字节码不同的是，Dalvik 字节码是基于寄存器的。在实际操作中，Java 源程序首先需要编译成 Java 字节码（class 文件），再经过 dx 工具转换为 Dalvik 字节码。

3.3.1 Dalvik 字节码总体设计

Dalvik VM 基于寄存器设计，Java 字节码转换为 Dalvik 字节码时，方法调用栈已经确定，其中明确指定使用寄存器个数以及额外的其他数据，如程序计数器 PC，引用另外一个 Dex 文件等。Dalvik 字节码可以使用的虚拟寄存器个数可达 65 536 个，每个寄存器有 32 位，以相邻的两个寄存器表示 64 位的数据。最终所有的寄存器会被映射到物理寄存器上，这将发挥出 RISC 架构寄存器多的优势，也是初期为 ARM 平台设计的产物。

字节码的操作码是 1B，操作数以 16 位为一个单元，类、方法或字符串等常量以使用常量池来引用。和 Java 字节码不同的是，Dalvik 字节码是以小端存储，并且指令的类型没有限制。32 位的寄存器中可以存储任意类型的数据，如 float 或 int。其设计需要满足以下几个原则。

（1）参数顺序是：目的寄存器，源寄存器。

（2）明确指定操作码类型。如通用类型的 64 位操作码通常有-wide 后缀，其他有-float、-double 等。另外还有些特殊后缀，如 move 指令，在操作数寄存器号用 4 位表示时，指令是 move vA, vB；但在操作数寄存器号是用 16 位表示时，指令是 move/16 vAAAA,

vBBBB。

3.3.2 Dalvik 字节码指令格式

表 3.4 展示了部分 Dalvik 字节码的指令格式。具体查看 Google 官方文档。

表 3.4 Dalvik 字节码指令格式

格　式	ID	语　法
₵₵|op	10x	Op
B|A|op	12x	Op vA,vB
AA|op	11x	Op vAA
	10t	Op +AA
AA|op BBBB	22x	Op vAA, vBBBB
AA|op CC|BB	23x	Op vAA,vBB,vCC
B|A|op CCCC	22t	Op vA,vB,+CCCC
B|A|op CCCC G|F|E|D	35c	[B=5] op {vD,vE,vF,vG}, type@CCCC [B=5] op {vD,vE,vF}, type@CCCC [B=5] op {vD,vE}, type@CCCC [B=5] op {vD}, type@CCCC [B=5] op {}, type@CCCC

对于表 3.4,第一列是指令格式,其包括 1 至多个单词,单词之间以空格相隔,每个单词代表了 16 位。单词中的每个字符代表 4 位(1 个单词最多有 4 字符),字符以从高至低的顺序读取,同时加上"|"号以便读取。指令中的大写字母表示在该位置有数或寄存器。"op"表明了该位置有 8 位操作码以及"exop"表示这是一个扩展的 16 位操作码。"Ø"则表明在该位置的应该为 0。大多数情况下,代码都是从左到右读取,从低到高在代码单元中排列。以指令格式"B|A|op CCCC"为例。该指令由两个 16 位单元组成,第一个 16 位为低 8 位的操作码和高 8 位的两个操作数,第二个 16 位则是一个 16 位的操作数。

第二列的 ID 号是指令格式的缩写。大部分的 ID 都有两个数字及之后的一个字母组成。首个数字指明了指令的长度(以 16 位为一个单位)。第二个数字表示的是指令最多能有多少个寄存器(有些指令的寄存器个数是不一定的)。如果第二位为字母"r",则表示编码了一组寄存器。最后一位的字符指示了额外数据的属性,详细说明见表 3.5。

表 3.5 额外数据属性说明

单　词	位　大　小	含　义
b	8	1 个字节的有符号立即数
c	16, 32	常量池索引
f	16	接口常量(只在静态链接中使用)
h	16	32 或 64 有符号立即数中的高 16 位,低位全为 0
i	32	有符号整型立即数,或者 32 位的浮点

续表

单 词	位 大 小	含 义
l	64	有符号长整型，或者 64 位的双精度
m	16	方法常量，只在静态链接中有使用
n	4	有符号 4 位立即数
s	16	有符号短整型立即数
t	8，16，32	跳转
x	0	没有额外的数据

比如，“21t”表示该指令长为 2(32 位)、有一个寄存器引用以及包含一个分支跳转。

Dalvik 字节码目前共有 226 条指令，每一条指令有一个对应的索引号。在具体实现中，Dalvik VM 会维护一张 Hash 表，以指令操作码为索引。Hash 表中的元素是每一条 Dalvik 字节码指令相等价的汇编或 C 语言实现。解释器取指后解码获得的字节码号就是这个作用。

以“move-wide/from16 vAA，vBBBB”指令为例：

(1) move 是指令最基本的操作码，说明了这个指令的功能；

(2) wide 是指令名称后缀，说明指令需要操纵 64 位的数据；

(3) from16 是指令功能后缀，说明源寄存器号需要 16 位表示；

(4) vAA 是目的寄存器，寄存器号必须在 v0～v255 之间；

(5) vBBBB 是源寄存器，寄存器号必须在 v0～v65535 之间；

(6) 该指令的指令格式是 22x，其长度是两个 16 位，即 4B；共使用了两个寄存器，并且没有额外的数据；

(7) 指令字节码号是 0x05(根据指令表)。

3.4 Odex 文件简介

Android 设备在使用的过程中会为目标程序生成一个后缀为.odex 的“优化文件”，其作用就是取代原有的可执行文件，以加快程序的启动执行速度，同时这些“优化文件”将永久地保存在手机的系统缓存中，但随着使用时间的增长，手机内部缓存中会产生大量的“缓冲文件”，这对于手机内存容量较小的用户来说，这些“优化文件”非但不能提高手机的运行速度，甚至还会使手机由于内存资源过低而出现各种问题。

因此，在很多手机技术论坛上，有大量的技术贴讲授如何为手机“减负”，想必有些读者一定有印象，在多种“减负”手段中，有一招就是删除系统内核中 data/dalvik-cache 文件夹下的文件，在一般情况下都可以释放出几十兆字节甚至上百兆字节的内存空间，让手机一下子重获新生。事实上，所删除的文件就是前面提到的“优化文件”，但在删除这些文件的时候，我们是否考虑过这些文件都是用来干什么的？为什么可以随意删除它们？在下面的讲解中将回答这些问题。

3.4.1　什么是“优化文件”

前面讲到，Android 系统为了提高程序的启动以及运行速度，特别为目标应用生成了与之对应的“优化文件”。这些优化文件正是保存在前面提到的 data/dalvik-cache 文件夹中，它们就是俗称的 Odex 文件（Optimized-Dex），是对 Android 应用程序中所包含的 Dex 文件在内容上以及结构上进行优化改写而生成。在对 Dex 文件进行优化的过程中，主要对其中的类数据进行安全性检验以保证其不会威胁虚拟机的安全运行，以及结合当前平台硬件特性对程序源码进行优化以提高程序的执行速度。

3.4.2　Odex 文件结构

直观上 Odex 文件在 Dex 文件的原有的结构上进行了扩充，即在 Dex 文件前拼接了 Odex 文件头部信息，还在 Dex 文件尾部拼接了依赖库、寄存器映射关系以及类的哈希索引等辅助信息。其结构对比如图 3.3 所示。

下面再通过 Odex 文件的头部信息可以更好地了解一下 Odex 的文件结构以及各部分数据含义，表 3.6 为 Odex 文件头 DexOptHeader 在 Dexfile.h 文件中的定义。

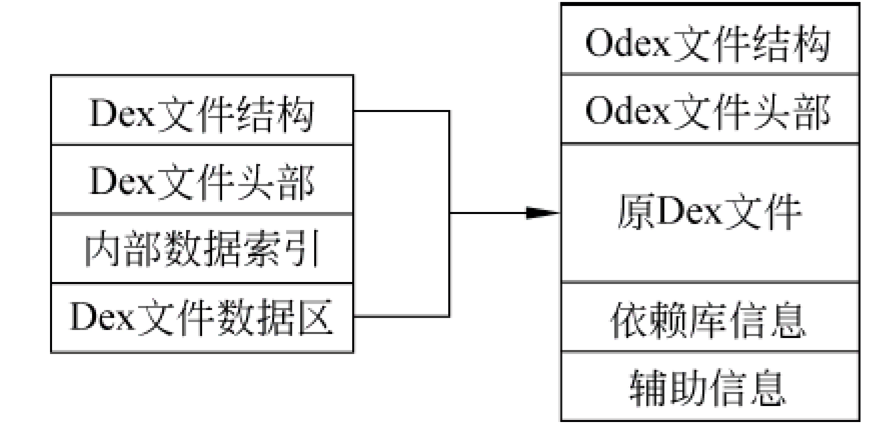


图 3.3　Dex 文件与 Odex 文件结构对比图

表 3.6　DexOptHeader 数据结构定义

变量类型	变量名称	描述
u1	Magic[8]	Odex 文件版本标识
u4	dexOffset	Dex 文件头偏移量
u4	dexLength	Dex 文件总长度
u4	depsOffset	Odex 文件依赖库列表偏移量
u4	depsLength	依赖库信息总长度
u4	optOffset	优化数据信息偏移量
u4	optLength	优化数据总长度
u4	flags	标识位
u4	checksum	文件校验和

表 3.6 中，DexOptHeader 结构中的 magic 字段与 DexHeader 结构中的 magic 字段类似，都是用于标识文件；dexOffset 字段表示原 Dex 文件起始位置的偏移量，实际上它就等于 DexOptHeader 结构体的大小 0x28；dexLength 字段表示 Dex 文件的总长度，通过这两个字段可以非常快速定位并读取 Dex 文件；depsOffset 字段表示依赖库起始的偏移量；depsLength 表示依赖库的总长度；optOffset 字段表示优化数据的起始偏移量；optLength 字段表示优化信息的总长度，而对于类加载机制非常关键的类索引信息就封装在这部分优化信息中；flags 字段为一个标识，其用于标示 Dalvik 虚拟机加载 Odex 文件时优化与验证

选项；checksum 字段为 Odex 文件的校验和。

通过这个头部信息，虚拟机可以非常高效地查找 Dex 文件中的各类信息，极大提高了执行效率。另外，Dalvik 虚拟机对 Dex 文件所进行的优化工作主要体现在依赖库和辅助信息两部分上，因此，下文将会对这两部分内容的功能进行介绍。

1. 依赖库信息

依赖库顾名思义，就是指该 Dex 文件所需要链接的本地函数库，Dalvik 虚拟机在程序执行前期通过优化机制将这部分整合到 Odex 文件中，可以在一定程度上提高程序的执行效率，表 3.7 为依赖库 Dependence 结构体定义。

表 3.7 Dependence 数据结构定义

变 量 类 型	变 量 名 称	描 述
u4	modWhen	时间戳
u4	crc	校验信息
u4	DALVIK_VM_BUILD	虚拟机版本号
u4	numDeps	依赖库个数
u4	len	Name 长度
u4	name[len]	依赖库名称
KSHA1DigestLen	signature	SHA-1 值

在表 3.7 中，modWhen 用来记录 Dex 文件优化前的时间戳，crc 为 Dex 文件优化前的 crc 校验值，DALVIK_VM_BUILD 值表示的是虚拟机版本号；不同版本的 Android 系统定义不同，例如：Android 2.2.3 为 19、Android 2.3 为 23 以及 Android 4.0.4 则为 27。numDeps 字段所代表的含义为该 Dex 文件的依赖库个数，其中 table 结构体的个数正是由 numDeps 决定的，也可以理解为每个依赖库都对应一个 table 结构体对象，在该结构体中，len 表示依赖库名称的长度、name 为依赖库名以及 signature 表示 SHA-1 签名。

2. 类索引信息

类索引信息的建立是优化机制的重要工作之一，在该索引表中，优化机制为 Dex 文件中的每一个类配置了一个 table 结构体对象，在这个对象中记录了类描述符哈希值、类描述符在 Dex 文件中偏移地址以及类定义区的偏移地址，类加载机制通过这些信息可以非常快速地定位类资源地址并加载类。同时，通过哈希查找的方式大大提高了类加载机制的查找效率，表 3.8 为 DexClassLookup 结构体定义。

表 3.8 中，numEntries 是一个比较特别的数目，它虽然表示的是表的项数，但实际上这个数值是通过 dexRoundUpPower2()函数生成的。

点拨 dexRoundUpPower2()函数是源自斯坦福大学的一个算法——用于求比一个数大的最小的 2 的整数次幂，例如，当数为 6 时，该算法计算得到 8。这样做的结果会比 Dex 文件中类的数量大，但好处是降低了哈希冲突率。

表 3.8　DexClassLookup 数据结构定义

变量类型	变量名称	描　　述
int	size	表大小
int	numEntries	表项入口数量
u4	classDescriptorHash	类描述符的哈希值
u4	classDescriptorOffset	Dex 文件中该类描述符的偏移位置
u4	classDefOffset	Dex 文件中该类定义偏移位置

3.4.3　Odex 文件加速系统运行速度

熟悉 Java 语言的读者应该知道，标准的 Java 虚拟机在执行一个应用程序前，会对程序中的类数据进行验证并优化，但是这种标准的验证优化机制并不会对程序源码数据造成永久性改变。因此，标准的 Java 虚拟机每每在执行程序之前都要反复做这项工作，在效率上有一定损失，但是鉴于大多数的 Java 虚拟机都是运行在 PC 之上，而当今 PC 的性能又都普遍较高，在效率上的这点损失就显得微不足道了。但是，我们都知道 Android 系统主要是运行在我们熟悉的各类移动设备之上，例如：平板电脑、手机、GPS 导航设备等，而移动设备往往具有处理器计算能力低下以及内存资源紧张等硬件限制，其整体性能远不及当今家用 PC，想要在嵌入式设备上流畅运行标准 Java 虚拟机几乎是一件不可能的事情。因此，Dalvik 虚拟机的开发者通过使用多种手段以提高程序运行速度，并减少不必要的操作，不但完全兼容标准 Java 语言，而且还尽可能地保证了用户使用感受。

在多种用于提高程序执行效率的手段中，将应用程序数据验证与优化工作的前置是 Dalvik 虚拟机功能架构设计上的一个创新，是区别于其他 Java 虚拟机的重要特点之一。前面提到，虚拟机在运行目标程序之前需要对其中的数据进行验证优化，随后再对优化过的数据进行执行。但是对于运行 Android 系统的嵌入式设备来说，大量的验证与优化工作可能会对系统造成沉重的负担，那么有什么办法可以避免重复多次地对程序数据进行验证优化呢？最为直接的办法就是长期保留目标程序的优化数据，而我们经常提及的 Odex 文件实际上就是用来保存这些优化数据，这样做的好处是当程序再次运行时，系统将首先判断目标程序是否存在对应的 Odex 文件，如果存在将跳过验证与优化这一步，直接运行 Odex 文件中的数据。这就是为什么在 Android 系统中，程序第一次启动的时间相对较长，而多出来的这部分时间正是用来为目标程序生成与之对应的 Odex 文件并将该文件保存在 cache 中，大大降低了程序再次启动的时间与性能消耗。

关于 Dex 文件验证优化机制的技术原理以及相关的实现细节，请参阅本系列丛书第二卷第 1 章。

3.4.4　手机“减负”问题再讨论

再回到为手机“减负”的问题上，简单讨论一下 cache 中的 Odex 文件到底值不值得删除。在理论上，删除 Odex 文件不会造成任何的系统故障，其唯一可能的影响就是：由于缺失对应的 Odex 文件，虚拟机将会再一次对程序进行优化并为之生成对应的 Odex 文件。说

到这里，不禁有读者要问，这样做似乎没有任何好处啊！即便删掉了 Odex 文件，但是虚拟机仍然还会再次生成，不仅没有达到节省资源的目的，而且又一次损失了时间(用于再次对 Dex 文件进行验证与优化)，实在得不偿失。

其实可以换个角度来看待这个问题，想必很多读者的手机中或多或少都安装了各类应用，出于各种原因，其中的部分应用会被用户卸载删除，但是这些被卸载的应用程序所对应的 Odex 文件有可能没有被删除，仍然保存在 cache 中占据着设备那可怜的内存资源。因此，对于这类 Odex 文件，应该坚决把它们删除，收复它们占用的内存，这样不就真的做到了“减负”的目的了么。总结来说，就 Odex 文件的删除原则来说主要归纳为以下三点：

(1) 对于常用的应用，其对应的 Odex 文件不能被删除。

(2) 对于不太常用的应用但又不准备卸载，可以选择删除所对应的 Odex 文件。

(3) 对于已经被卸载的应用，其对应的 Odex 文件要坚决删除毫不姑息。

Odex 文件的开发与设置，实际上是对设备存储空间的一种牺牲，而这种牺牲却换取了更好的计算性能，使用户的使用感受得到显著提高。依据前面提到的三点建议，相信读者们都能或多或少地回收一些宝贵的内存资源，然而随着应用软件的高速发展，对设备的性能以及资源的需求越来越大，在此只能盼望硬件快速发展，早日摆脱硬件带来的局限。

小结

Dex 文件作为 Android 系统的可执行文件，封装了应用程序的全部操作指令以及程序数据，其于 Dalvik 虚拟机的重要性不言而喻。本章主要对 Dex 文件中所涉及的各个数据结构的函数定义进行了分析并结合一个 Dex 文件实例进行讲解，同时还对 Dalvik 字节码进行了全面的介绍，主要包括字节码设计、字节码格式等内容。最后，本章对 Dex 文件的优化产物 Odex 文件功能原理与实际应用进行了简单的介绍。希望读者通过本章的学习能够对 Dex 文件有了一个更深层次的认识。

第 4 章 系统工具

本章主要内容

✍如何对 Dex 文件进行反编译?

✍Dex 文件及 apk 对系统包系统类及系统函数存在着怎样的依赖?

✍dexlist 工具是怎样获取并列举类中的方法的?

✍Dex 文件是如何实现优化和验证的?

✍以 dvz 的方式启动一个进程如何实现?

在 Dalvik 虚拟机中存在着大量的系统工具,完成虚拟机相关系统操作的同时,用户也可以利用这些工具分析虚拟机内部机制,Dex 文件作为虚拟机运行过程中的一个核心文件,分析理解 Dex 文件对于认识虚拟机内部机制,提高虚拟机中程序的运行速度有很大帮助。此外,一些源码分析工具也帮助程序员解决一些调试过程中棘手的问题,熟练运用工具解决编码中出现的一些内存泄漏及堆栈溢出的问题也十分有效。

4.1 本章概述

对于操作系统来说,除了平时使用的功能以外,都有可供用户使用的工具,丰富系统的功能,同时也可以由用户完成对系统内核的一些分析操作。在 Dalvik 虚拟机中也有很多工具,你了解它们的功能吗?为了更好地理解 Dalvik 虚拟机附带的工具,了解系统工具完成的功能。在第 3 章对 Dex 文件进行介绍的基础上,本章结合 Dalvik 的系统工具,围绕 Dex 文件的相关操作展开,使用 Dalvik 中的系统工具对 Dex 文件进行分析,提炼 Dex 文件中包含的信息,进行相关的优化,分析 Dex 文件运行所依赖的相关配置,能够很好地完成对 Dalvik 中内部机制的理解。此外,系统工具还提供了对 Android 程序源码进行调试的功能,可以有效地提高程序运行的效率。

本章主要介绍 Dex 文件的逆向工具,dexdump 工具主要完成对 Android 程序的反编译,用于逆向分析在 apk 程序生成过程中的 Dex 文件。dexdeps 工具是 Dex 文件的依赖工具,分析程序使用的依赖包信息。dexlist 工具列出一个 Dex 文件中所有具体类中的所有方法;Dex 文件优化工具 dexopt 主要完成对 Dex 文件进行优化,提高文件的载入速度。此外,启动进程的 dvz 工具完成对进程启动。

4.2 dexdump 工具

4.2.1 dexdump 工具简介

逆向工程作为现在一个比较热门的领域，针对不同类型的程序的逆向工具也很多，很多也非常成熟，dexdump 作为一个 Android 平台下的反编译工具，用于获取文件的检验和、文件摘要、文件头信息、布局、寄存器映射等信息，通过对生成的文件进行分析，可以得到 Android 程序源码中的一些信息，在源码未知的情况下，在知道功能同时可以对目标程序进行分析，同时这也带来了代码不安全的问题，程序的漏洞被利用这样一个信息安全问题，使用这个工具也可以对自己的程序的安全性进行分析和验证。将已经打包完成的 apk 程序解压，分析包中的 Dex 文件，Dex 文件相关内容已在 3.2 节进行了介绍。对 Dex 文件进行反编译，得到程序代码的相关信息便于分析。dexdump 工具的源码位置位于 Android_dalvik_source\dexdump，读者有兴趣可以参阅学习。

4.2.2 dexdump 工具使用方法

dexdump 工具作为一个 Android 平台下的反编译工具，其源文件是 Dex 文件，Dex 文件存在于 apk 安装包中。对 apk 文件进行处理，apk 文件是 Android 程序打包后的安装程序，在安装了 Android SDK 开发包的 Eclipse 开发环境下可以很方便将程序打包。解压缩 apk 文件后，得到待分析的 Dex 文件。

其中文件后缀为 dex 的即为所需的 Dex 文件，其他文件如 AndroidMainfest.xml 为主体框架的配置文件。

本节将介绍两种使用 dexdump 工具反编译 Dex 文件的方法，一种是不启动 Android 模拟器直接对 Dex 文件进行反编译，另一种方法是启动 Android 模拟器反编译 Dex 文件。

第一种方法，直接在命令行界面下调用 dexdump 工具对目标 dex 文件进行反编译，无须启动 Android 模拟器，具体的实现步骤如下。

第一步：配置 Android 模拟器相关的环境变量

使用的操作系统为 Ubuntu 10.04，通过之前几章的介绍相信读者已经对 Linux 操作系统下的相关操作有了一定的了解，相关的命令行操作可通过 Ctrl＋Alt＋T 启动终端来完成，首先，通过调用 cd 命令完成进入源码目标路径的转换。本例中 Android 源码所在的文件目录为 Androidsource/Android4.04，相关的配置过程如图 4.1 所示，配置环境变量的主要命令如下：

```
build/envsetup.sh
lunch 1
```

第二步：调用 dexdump 命令对 Dex 文件进行反编译

以 dexdump 的-d 参数进行反编译为例，源 Dex 文件为 fibo.dex，存放在 dextest 文件夹下，输出文件为 classtest.txt，调用命令 dexdump -d fibo.dex > classtest.txt，其中 classtest.txt 为反编译后的输出文件，获得输出文件的代码块的信息，核心反编译命令：dexdump -d fibo.dex > classtest.txt，相关的命令执行过程如图 4.2 所示。

```
cheng@cheng-laptop: ~/androidsource/android4.0.4
File Edit View Terminal Help
cheng@cheng-laptop:~$ cd androidsource
cheng@cheng-laptop:~/androidsource$ cd android4.0.4
cheng@cheng-laptop:~/androidsource/android4.0.4$ . build/envsetup.sh
including device/moto/stingray/vendorsetup.sh
including device/moto/wingray/vendorsetup.sh
including device/samsung/crespo4g/vendorsetup.sh
including device/samsung/crespo/vendorsetup.sh
including device/samsung/maguro/vendorsetup.sh
including device/samsung/torospr/vendorsetup.sh
including device/samsung/toro/vendorsetup.sh
including device/samsung/tuna/vendorsetup.sh
including device/ti/panda/vendorsetup.sh
including sdk/bash_completion/adb.bash
cheng@cheng-laptop:~/androidsource/android4.0.4$ lunch 1

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.0.4
TARGET_PRODUCT=full
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
```

图 4.1　配置模拟器启动的环境变量

```
cheng@cheng-laptop: ~/dextest
File Edit View Terminal Help
including device/ti/panda/vendorsetup.sh
including sdk/bash_completion/adb.bash
cheng@cheng-laptop:~/androidsource/android4.0.4$ lunch 1

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.0.4
TARGET_PRODUCT=full
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=IMM76L
============================================

cheng@cheng-laptop:~/androidsource/android4.0.4$ cd ..
cheng@cheng-laptop:~/androidsource$ cd ..
cheng@cheng-laptop:~$ cd dextest
cheng@cheng-laptop:~/dextest$ dexdump -d fibo.dex>classtest.txt
cheng@cheng-laptop:~/dextest$
```

图 4.2　执行 dexdump 命令对 Dex 文件进行反编译

其中,-d 操作对代码进行拆分操作,-c 操作对校验和进行检验操作。-c,-f,-h 操作与-d 操作产生的效果类似。关于 dexdump 反编译中使用的命令和相关参数以及其具体含义如下:

```
dexdump: [-c] [-d] [-f] [-h] [-i] [-l layout] [-m] [-t tempfile] dexfile...
-c : verify checksum and exit
-d : disassemble code sections
```

```
-f : display summary information from file header
-h : display file header details
-i : ignore checksum failures
-l : output layout, either 'plain' or 'xml'
-m : dump register maps (and nothing else)
-t : temp file name (defaults to /sdcard/dex-temp-*)
```

作者总结出来的使用中方便记忆相关操作命令的方法,如表 4.1 所示。

表 4.1 dexdump 工具反编译命令

命　令	功能及记忆方法
-c	检查检验和并退出,checksum 首字母 c
-d	拆分代码区域,disassemble 首字母 d
-f	从文件头中显示的摘要信息,file 首字母 f
-h	显示文件头的详细信息,header 首字母 h
-i	忽略检验和失败,ingore 首字母 i
-l	输出布局以 plain 或者 xml 形式,layout 首字母 l
-m	转储寄存器映射,maps 首字母 m

第三步:分析反编译生成的文件

生成 fibo.dex 文件的 Java 程序源文件 Fibonacci.java 如代码清单 4.1 所示。

代码清单 4.1 手动编写 Fibonacci.java 源代码

```
import java.lang.System;
public class Fibonacci {
    private long fib_java(int num)
    {
        if (num==0){
            return num;
        }
        else
        {
            return fib_java(num-1)+fib_java(num-2);
        }
    }
    public static void main(String args[])
{
        Fibonacci fib=new Fibonacci();
        fib.fib_java(35);
        System.out.println("test Fibonacci");

}
}
```

源文件 Fibonacci.java 主要完成在自定义函数中计算斐波那契数列的功能。在主函数中完成实例化 Fibonacci 类的一个对象,并调用斐波那契数列的计算函数,完成相关功能,最后打印出"test Fibonacci"。

在第二步中对 fibo.dex 进行反编译,使用 dexdump -d fibo.dex > classtest.txt 命令,得到 classtest.txt 文件,如代码清单 4.2 所示。

代码清单 4.2 反编译结果 classtest.txt 代码

```
    Processing 'fibo.dex'...
    Opened 'fibo.dex', dex version '035'
    Class #0                  -
      Class descriptor        : 'LFibonacci;'
      Access flags            : 0x0001 (PUBLIC)
      Superclass              : 'Ljava/lang/Object;'
      Interfaces              -
      Static fields           -
      Instance fields         -
      Direct methods          -
        #0                    : (in LFibonacci;)
          name                : '<init>'
          type                : '()V'
          access              : 0x10001 (PUBLIC CONSTRUCTOR)
          code                -
          registers           : 1
          ins                 : 1       outs            : 1
      insns size              : 4 16-bit code units
00015c:                       |[00015c] Fibonacci.<init>:()V
00016c: 7010 0400 0000         |0000: invoke-direct {v0}, Ljava/lang/Object;.<init>:()V // method
@0004
000172: 0e00                  |0003: return-void
      Catches                 : (none)
      Positions               :
      0x0000 line=2
      Locals                  :
        0x0000 - 0x0004 reg=0 this LFibonacci;

    #1                        : (in LFibonacci;)
      name                    : 'fib_java'
      type                    : '(I)J'
      access                  : 0x0002 (PRIVATE)
      code                    -
      registers               : 6
      ins                     : 2
      outs                    : 2
      insns size              : 20 16-bit code units
000174:                       |[000174] Fibonacci.fib_java:(I)J
```

```
000184: 3905 0400                |0000: if-nez v5, 0004 // +0004
000188: 8150                     |0002: int-to-long v0, v5
00018a: 1000|0003: return-wide v0
00018c: 1210                     |0004: const/4 v0, #int 1 // #1
00018e: 9100 0500                |0005: sub-int v0, v5, v0
000192: 7020 0100 0400           |0007: invoke-direct {v4, v0}, LFibonacci;.fib_java:(I)J // method@0001
000198: 0b00                     |000a: move-result-wide v0
00019a: 1222                     |000b: const/4 v2, #int 2 // #2
00019c: 9102 0502                |000c: sub-int v2, v5, v2
0001a0: 7020 0100 2400           |000e: invoke-direct {v4, v2}, LFibonacci;.fib_java:(I)J // method@0001
0001a6: 0b02                     |0011: move-result-wide v2
0001a8: bb20                     |0012: add-long/2addr v0, v2
0001aa: 28f0                     |0013: goto 0003 // -0010
      catches                    : (none)
      positions                  :
        0x0000 line=5
        0x0002 line=6
        0x0003 line=10
      locals          :
        0x0000 - 0x0014 reg=4 this LFibonacci;

    #2                           : (in LFibonacci;)
      Name                       : 'main'
      Type                       : '([Ljava/lang/String;)V'
      Access                     : 0x0009 (PUBLIC STATIC)
      Code                       -
      Registers                  : 3
      Ins                        : 1
      Outs                       : 2
      insns size                 : 18 16-bit code units
0001ac:                          |[0001ac] Fibonacci.main:([Ljava/lang/String;)V
0001bc: 2200 0200                |0000: new-instance v0, LFibonacci; // type@0002
0001c0: 7010 0000 0000           |0002: invoke-direct {v0}, LFibonacci;.<init>:()V
// method@0000
0001c6: 1301 2300                |0005: const/16 v1, #int 35 // #23
0001ca: 7020 0100 1000           |0007: invoke-direct {v0, v1}, LFibonacci;.fib_java:(I)J // method@0001
0001d0: 6200 0000                |000a: sget-object v0, Ljava/lang/System;.out:Ljava/
io/PrintStream; // field@0000
0001d4: 1a01 1100                |000c: const-string v1, "test Fibonacci" // string@0011
0001d8: 6e20 0300 1000           |000e: invoke-virtual {v0, v1}, Ljava/io/PrintStream;
.println:(Ljava/lang/String;)V // method@0003
0001de: 0e00                     |0011: return-void
      Catches                    : (none)
      Positions                  :
        0x0000 line=15
```

```
        0x0005 line=16
        0x000a line=17
        0x0011 line=20
      locals                  :

  Virtual methods             -
  source_file_idx             : 1 (Fibonacci.java)
```

点拨　输出的 txt 文件可能比较大，建议使用 Vim 打开，以免在分析时出现程序没有响应的情况。

反编译生成的文件中类描述中体现了类的名字 Fibonacci，访问标志是 public 的，紧接着＃0 开始是 Fibonacci 的一个初始化函数，展现了寄存器的数量为 1，输入输出的数量均为 1，指令集大小为 4 个 16 位的代码单元。

＃1 开始是 Fibonacci 类中的另外一个函数 fib_java，是 private 的。对应的代码段如下：

```
000174:                                 |[000174] Fibonacci.fib_java:(I)J
000184: 3905 0400                       |0000: if-nez v5, 0004 // +0004
000188: 8150                            |0002: int-to-long v0, v5
00018a: 1000                            |0003: return-wide v0
00018c: 1210                            |0004: const/4 v0, #int 1 // #1
00018e: 9100 0500                       |0005: sub-int v0, v5, v0
000192: 7020 0100 0400                  |0007: invoke-direct {v4, v0},
LFibonacci;.fib_java:(I)J // method@0001
000198: 0b00                            |000a: move-result-wide v0
00019a: 1222                            |000b: const/4 v2, #int 2 // #2
00019c: 9102 0502                       |000c: sub-int v2, v5, v2
0001a0: 7020 0100 2400                  |000e: invoke-direct {v4, v2},
LFibonacci;.fib_java:(I)J // method@0001
0001a6: 0b02                            |0011: move-result-wide v2
0001a8: bb20                            |0012: add-long/2addr v0, v2
0001aa: 28f0                            |0013: goto 0003 // -0010
```

将源文件和反编译后的文件进行对比分析，上述代码分别执行的是源代码中的第 5，6，10 行代码的内容。其完成的功能是对数值的判断，根据不同结果返回。

```
0001d0: 6200 0000                       |000a: sget-object v0, Ljava/lang/System;
.out:Ljava/io/PrintStream; // field@0000
0001d4: 1a01 1100                       |000c: const-string v1, "test Fibonacci"
// string@0011
```

不难看出上述代码完成的是一个打印语句的操作，调用 java/lang/System;. out:Ljava/io/PrintStream 包里的打印输出流函数完成对“test Fibonacci”语句的打印，在源码中对应的语句是 System. out. println("test Fibonacci")；

“0001de：0e00　0011：return-void”在程序的第 20 行大括号结束，返回出数值，类型为 void。而代码中的第 18 行和 19 行为空白行，在反编译过程中未产生任何内容。

上述代码中获得的是类的头信息，类的描述，访问标志，超类，接口，静态数据，实例数据，直接方法，虚方法。

类似地，可以调用反编译 Dex 文件校验和操作，并进行分析，调用 dexdump －f fibo.dex ＞ filesummary.txt 将生成文件摘要信息，由于篇幅原因，只选取其中一部分进行分析，选取摘要文件 filesummary.txt 的一部分进行分析。相关反编译代码如代码清单 4.3 所示。

代码清单 4.3　反编译部分代码

```
Processing 'fibo.dex'...
Opened 'fibo.dex', dex version '035'
dex file header:
Magic                        : 'dex\n035\0'
Checksum                     : 01987660
Signature                    : 3bb4...e957
file_size                    : 908
header_size                  : 112
link_size                    : 0
link_off                     : 0 (0x000000)
string_ids_size              : 18
string_ids_off               : 112 (0x000070)
type_ids_size                : 9
type_ids_off                 : 184 (0x0000b8)
field_ids_size               : 1
field_ids_off                : 268 (0x00010c)
method_ids_size              : 5
method_ids_off               : 276 (0x000114)
class_defs_size              : 1
class_defs_off               : 316 (0x00013c)
data_size                    : 560
data_off                     : 348 (0x00015c)

Class #0                     -
  Class descriptor           : 'LFibonacci;'
  Access flags               : 0x0001 (PUBLIC)
  Superclass                 : 'Ljava/lang/Object;'
  Interfaces                 -
  Static fields              -
  Instance fields            -
  Direct methods             -
    #0                       : (in LFibonacci;)
      Name                   : '<init>'
      Type                   : '()V'
      Access                 : 0x10001 (PUBLIC CONSTRUCTOR)
      Code                   -
      Registers              : 1
      Ins                    : 1
```

```
      Outs                   : 1
      insns size             : 4 16-bit code units
      Catches                : (none)
      Positions              :
        0x0000 line=2
      Locals                 :
        0x0000 - 0x0004 reg=0 this LFibonacci;
    #1                       : (in LFibonacci;)
      Name                   : 'fib_java'
      Type                   : '(I)J'
      Access                 : 0x0002 (PRIVATE)
      Code                   -
      Registers              : 6
      Ins                    : 2
      Outs                   : 2
      insns size             : 20 16-bit code units
      Catches                : (none)
      Positions              :
        0x0000 line=5
        0x0002 line=6
        0x0003 line=10
      Locals                 :
        0x0000 - 0x0014 reg=4 this LFibonacci;
    #2                       : (in LFibonacci;)
      Name                   : 'main'
      Type                   : '([Ljava/lang/String;)V'
      Access                 : 0x0009 (PUBLIC STATIC)
      Code                   -
      Registers              : 3
      Ins                    : 1
      Outs                   : 2
      insns size             : 18 16-bit code units
      Catches                : (none)
      Positions              :
        0x0000 line=15
        0x0005 line=16
        0x000a line=17
        0x0011 line=20
      Locals                 :
  Virtual methods            -
  source_file_idx            : 1 (Fibonacci.java)
```

其中,Dex 文件头包括检验和、签名、文件大小、头大小、字符串、类型、区域、方法、类相关的大小和偏移量。类信息包括类的描述,访问标志位,超类,静态区域(名字、类型、访问权限),实例区域(名字、类型、访问权限),直接方法(名字、类型、访问权限、代码、寄存器、指令、

输出、指令集大小位置等信息），虚方法等，这些在第3章中已经做过了详细介绍。

从反编译的分析结果不难看出，反编译后的文件可以较清晰地展现出类的信息，可以得到类里函数的功能，甚至是一些函数参数的信息，但更多的信息还是与 Dex 文件结构和字节码结构有关，深入分析还需要对 Dex 文件结构有深入的了解。从反编译的结果来看，程序的安全性不高，通过反编译 Dex 文件可以获得类的相关信息，这也提醒了程序员在编写代码过程中，尤其是像 Android 这种代码开源的操作系统，需要对代码进行保护，以免程序的漏洞被利用。

第二种方法，启动 Android 模拟器，在模拟器中使用 dexdump 对 Dex 文件进行反编译。

第一步：启动 Android 模拟器

启动 Android 模拟器，在上述配置相关环境变量的基础上，加入一步 emulator 命令，启动模拟器。配置及启动命令如下：

(1) . build/envsetup. sh；

(2) lunch 1；

(3) Emulator。

(1)和(2)为配置环境变量命令，(3)为启动模拟器命令，相关实现过程如图 4.3 所示。

```
cheng@cheng-laptop: ~/androidsource/android4.0.4
File Edit View Terminal Help
including device/samsung/maguro/vendorsetup.sh
including device/samsung/torospr/vendorsetup.sh
including device/samsung/toro/vendorsetup.sh
including device/samsung/tuna/vendorsetup.sh
including device/ti/panda/vendorsetup.sh
including sdk/bash_completion/adb.bash
cheng@cheng-laptop:~/androidsource/android4.0.4$ lunch 1

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.0.4
TARGET_PRODUCT=full
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=IMM76L
============================================

cheng@cheng-laptop:~/androidsource/android4.0.4$ emulator
```

图 4.3 执行 dexdump 命令对 Dex 文件进行反编译

第二步：创建 SD 卡并将 Dex 文件传入 SD 卡中

Android 模拟器创建一个 SD 卡，首先将路径转换到工作目录，转换目录命令 cd ＄WORKDIR。其中，＄WORKDIR 代表工作目录。本例中使用的工作目录为 Android 虚拟机的源码目录。创建命令为 mksdcard 128M sdcard. img ＃，创建了一个名为 sdcard. img 的存储卡，大小为 128MB。创建 SD 成功后，将要查看的 Dex 文件使用 adb push 的方式上传到模拟器中，创建一个 128MB 的 Android 模拟器使用的存储卡，具体的实现流程如图 4.4 所示。

其中在此过程中，可能会出现“failed to copy ‘classes. dex’ to ‘/sdcard/classes. dex’：

```
cheng@cheng-laptop:~/桌面/dextest$ adb push classes.dex /sdcard/
failed to copy 'classes.dex' to '/sdcard//classes.dex': Read-only file system
cheng@cheng-laptop:~/桌面/dextest$ adb shell mount -o remount rw /
cheng@cheng-laptop:~/桌面/dextest$ adb push classes.dex /sdcard/
2723 KB/s (4585428 bytes in 1.644s)
cheng@cheng-laptop:~/桌面/dextest$
```

图 4.4　创建 SD 卡并将 Dex 文件放入存储卡中过程

Read-only file system "的错误提示，使用挂载相关命令解决此问题，命令如下：adb shell mount -o remount rw /。

点拨　在将文件传入 SD 卡的过程中，SD 卡只需创建一次即可。

第三步：执行 dexdump 命令反编译 Dex 文件

然后通过 adb shell 登录，找到要查看的 Dex 文件，执行 dexdump xxx. dex。由于在 Android 模拟器运行 dexdump 工具对 SD 卡中的 Dex 文件进行反编译速度相对较慢，-d 操作生成的文件接近 100MB，因为使用方法类似，本节以对 Dex 反编译校验和操作进行分析为例介绍，读者可以自己进行举一反三。具体的命令如下：执行 dexdump 命令进行操作 adb shell dexdump -c /sdcard/classes. dex > checksum. txt，对代码块的分析命令类似 adb shell dexdump -d /sdcard/classes. dex > classes20130711. txt，读者有兴趣可以自己尝试。执行反编译命令如图 4.5 所示。

```
cheng@cheng-laptop: ~/androidsource/android4.0.4
File Edit View Terminal Help
including device/samsung/toro/vendorsetup.sh
including device/samsung/tuna/vendorsetup.sh
including device/ti/panda/vendorsetup.sh
including sdk/bash_completion/adb.bash
cheng@cheng-laptop:~/androidsource/android4.0.4$ lunch 1

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.0.4
TARGET_PRODUCT=full
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=IMM76L
============================================

cheng@cheng-laptop:~/androidsource/android4.0.4$ adb shell dexdump -c  /sdcard/c
lasses.dex > checksum.txt
cheng@cheng-laptop:~/androidsource/android4.0.4$
```

图 4.5　执行 dexdump 命令对 Dex 文件进行反编译

对反编译 Dex 文件后生成的文件的分析与方法一中的分析相同，已在 4.2.1 节进行了叙述，在此不再赘述。

4.3 dexdeps 工具

4.3.1 dexdeps 工具简介

dexdeps 工具主要完成对 Dex 文件、zip 文件、apk 文件的依赖信息的分析。此工具转储一个 Dex 文件使用但没有定义的字段和方法的列表，并列举出程序调用的不同的库函数。dexdeps 会寻找一个“classes. dex”项，用于分析文件的依赖信息。

4.3.2 dexdeps 工具使用方法

第一步，配置环境变量。

由于运行工具或是模拟器之前，都需要设置环境变量，前文已多次使用并给出了详细方法，后文不再对这一步多做说明。

第二步，启动 dexdeps 工具分析 zip 类型的文件。

(1) 对 Dex 文件进行分析。

首先，转换到 Dex 文件所在的目录。依赖分析相关命令如下：

dexdeps *. dex 或者将输出内容写入到一个 txt 文件中，相关命令如下：

```
dexdeps * .dex> * .txt
```

其中，*. dex 表示任意待分析的 Dex 文件，*. txt 表示输出的依赖文件，命令运行结果如图 4.6 所示。

```
cheng@cheng-laptop: ~/dextest
File  Edit  View  Terminal  Help
cheng@cheng-laptop:~/androidsource$ cd ..
cheng@cheng-laptop:~$ cd dextest
cheng@cheng-laptop:~/dextest$ dexdeps fibo.dex
<external file="fibo.dex">
  <package name="java.io">
    <class name="PrintStream">
      <method name="println" return="void">
        <parameter type="java.lang.String"/>
      </method>
    </class>
  </package>
  <package name="java.lang">
    <class name="Object">
      <constructor name="Object">
      </constructor>
    </class>
    <class name="String">
    </class>
    <class name="System">
      <field name="out" type="java.io.PrintStream"/>
    </class>
  </package>
</external>
cheng@cheng-laptop:~/dextest$
```

图 4.6 对 Dex 文件的依赖进行分析

(2) 分析 apk 文件。

依赖分析相关命令如下：

```
dexdeps * .apk> * .txt
```

其中，* .apk 表示任意待分析的 apk 文件，* .txt 表示输出的依赖文件，命令运行结果如图 4.7 所示。

```
cheng@cheng-laptop: ~/apk
File Edit View Terminal Help
    <class name="AtomicBoolean">
      <constructor name="AtomicBoolean">
      </constructor>
      <method name="get" return="boolean">
      </method>
      <method name="set" return="void">
        <parameter type="boolean"/>
      </method>
    </class>
    <class name="AtomicInteger">
      <constructor name="AtomicInteger">
        <parameter type="int"/>
      </constructor>
      <method name="getAndIncrement" return="int">
      </method>
    </class>
  </package>
  <package name="org.xmlpull.v1">
    <class name="XmlPullParserException">
    </class>
  </package>
</external>
cheng@cheng-laptop:~/apk$ dexdeps Fibonacci.apk > apkoutput.txt
cheng@cheng-laptop:~/apk$
```

图 4.7　对 apk 文件的依赖进行分析

第三步，分析生成的依赖文件

从 dexdeps 工具产生的依赖文件分析，相同的程序源码分别转换成 Dex 文件和 apk 文件，对应的依赖集差别很大，Dex 文件和 apk 文件对应的依赖集不同。对 Dex 文件的依赖关系文件进行分析，相关依赖关系如代码清单 4.4 所示。

代码清单 4.4　Dex 文件依赖代码

```
<external file= "fibo.dex">
  <package name= "java.io">
    <class name= "PrintStream">
      <method name= "println" return= "void">
        <parameter type= "java.lang.String"/>
      </method>
    </class>
  </package>
  <package name= "java.lang">
    <class name= "Object">
      <constructor name= "Object">
      </constructor>
    </class>
    <class name= "String">
    </class>
    <class name= "System">
```

```
        <field name="out" type="java.io.PrintStream"/>
      </class>
    </package>
  </external>
```

点拨 在本节中提到的依赖集是指程序运行过程中必须依赖的库文件以及包文件。

文件是以 XML 文件的形式输出的，可以看出分析的 Dex 文件的名字，列举出使用到的 Java 的包的名字，包里面具体类的名字，依赖类中方法的名字。本例中，使用到的包名为 java. io，类名为 PrintStream，具体的方法名字为 println，返回值 void 类型，方法对应的参数类型是 java. lang. String。还使用到了 java. lang 包的类和方法，类分别是 Object 和 System，其中 Object 类完成对象的相关操作，System 类完成流的输出。

不难看出，使用 dexdeps 工具生成的分析文件主要是对在程序源文件中调用的系统类和方法进行分析。在程序源码中 println()用到了上述类中的方法。

而相反地，分析 apk 文件产生的输出文件 apkoutput. txt，内容比较多。同样是完成一个简单的功能，apk 文件相对于 Dex 文件所需要的依赖包就多很多类和方法，apk 文件需要一些布局文件完成程序运行的相关操作，用到的很多依赖包都是 Android 环境下的包文件。由于生成的文件内容较多，选取一部分进行分析，相关代码如代码清单 4. 5 所示。

代码清单 4. 5 apkoutput. txt 源代码

```
<external file="Fibonacci.apk">
  <package name="Android.accessibilityservice">
    <class name="AccessibilityServiceInfo">
      <method name="getCanRetrieveWindowContent" return="boolean">
      </method>
      <method name="getDescription" return="java.lang.String">
      </method>
      <method name="getId" return="java.lang.String">
      </method>
      <method name="getResolveInfo" return="Android.content.pm.ResolveInfo">
      </method>
      <method name="getSettingsActivityName" return="java.lang.String">
      </method>
    </class>
  </package>
  <package name="Android.animation">
    <class name="ValueAnimator">
      <method name="getFrameDelay" return="long">
      </method>
    </class>
  </package>
```

上述代码中，首先显示了进行分析的 apk 文件名字，包名 Android 中的包 accessibilityservice，类名 AccessibilityServiceInfo，表示访问服务信息，以下依次列举出使用到的包中的方法名字和返回值。

4.4　dexlist 工具

4.4.1　dexlist 工具简介

dexlist 工具是一个通过解析 Dex 文件并反编译其中的类和方法，实现列出所有具体类中的一个或多个 Dex 文件中的所有方法的功能。在虚拟机启动时加载，为虚拟机的部分模块提供相关文件，主要由 Dalvik 虚拟机自身完成对该工具的调用，dexlist 工具源码位于 Android_dalvik_source\dexlist 目录下。

4.4.2　dexlist 工具使用说明

dexlist 工具是首先在主函数中调用 process()函数，实现启动一个 Dex 文件分析进程的操作，在函数中首先调用 Dex 文件打开、映射、分析等函数，其中，dexOpenAndMap()和 dexFileParse()这两个函数是类加载过程中非常核心的函数，在类加载章节中已经介绍过了，可以参见相关章节进行分析。整个工具的实现流程图如图 4.8 所示。

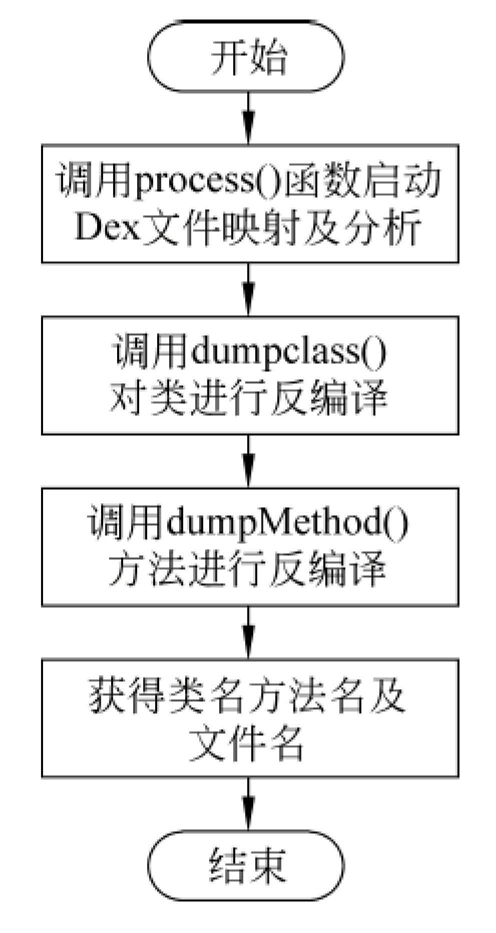


图 4.8　dexlist 工具实现流程

在 dexlist 工具主函数 main 函数中，首先，查找所有实例的完全限定的方法名，循环运行列表中的文件。在此过程中，usage()函数打印语句展现对应 Dex 文件结果，在此过程中调用 process()函数启动 Dex 文件分析进程。相关代码如代码清单 4.6 所示。

代码清单 4.6　dalvik/dexlist:main()源代码

```
int main(int argc, char*  const argv[])
{
    int result=0;
    int i;
/*查找所有实例的完全限定的方法名。这不是真正 dexlist 想要做的,但是在这里是容易这样做的。*/
    if (argc>3 && strcmp(argv[1], "--method")==0) {
        gParms.argCopy=strdup(argv[2]);
        char*  meth=strrchr(gParms.argCopy, '.');
        if (meth==NULL) {
            fprintf(stderr, "Expected package.Class.method\n");
            free(gParms.argCopy);
            return 2;
        }
        *meth='\0';
        gParms.classToFind=gParms.argCopy;
        gParms.methodToFind=meth+1;
        argv +=2;
        argc -=2;
```

```
    }

    if (argc < 2) {
        fprintf(stderr, "%s: no file specified\n", gProgName);
        usage();
        return 2;
    }
/* 运行列表中的文件。如果其中一个失败则继续，只返回一个失败到最后 */
    for (i=1; i < argc; i++)
        result |= process(argv[i]);

    free(gParms.argCopy);
    return result;
}
```

Dex 文件解析进程，打开 Dex 文件并映射到内存中，此时判断 Dex 文件是否打开和映射成功，如果映射不成功，释放 Dex 文件，如果 Dex 文件打开并映射成功，调用 dexFileParse()函数开始分析 Dex 文件，循环调用 dumpClass()函数对类信息进行获取，该函数需要两个参数，分别是 Dex 文件和 Dex 文件中定义的类的数量。对 Dex 文件进行解析。运行一个文件成功返回 0。相关代码如代码清单 4.7 所示。

代码清单 4.7 dalvik/dexlist:process()函数源代码

```
int process(const char* fileName)
{
    dexFile* pdexFile=NULL;
    MemMapping map;
    bool mapped=false;
    int result=-1;
    UnzipToFileResult utfr;

    utfr=dexOpenAndMap(fileName, NULL, &map, true);
    if (utfr != kUTFRSuccess) {
        if (utfr==kUTFRNoClassesdex) {
/* 没有 classes.dex 文件在 apk 中，假设成功。 */
            result=0;
            goto bail;
        }
        fprintf(stderr, "Unable to process '%s'\n", fileName);
        goto bail;
    }
    mapped=true;

    pdexFile=dexFileParse((u1*)map.addr, map.length, kdexParseDefault);
    if (pdexFile==NULL) {
        fprintf(stderr, "Warning: dex parse failed for '%s'\n", fileName);
```

```
            goto bail;
        }

        printf("# %s\n", fileName);

        int i;
        for (i=0; i < (int) pdexFile->pHeader->classDefsSize; i++) {
            dumpClass(pdexFile, i);
        }

        result=0;

    bail:
        if (mapped)
            sysReleaseShmem(&map);
        if (pdexFile != NULL)
            dexFileFree(pdexFile);
        return result;
    }
```

对 Dex 文件中的类中的直接方法和虚方法进行分析的函数 dumpClass()。首先调用 dexGetClassDef()函数从 Dex 文件中获得类的定义，调用 dexGetClassData()获得类数据，调用 dexReadAndVerifyClassData()函数读取 Dex 文件验证类的数据。判断类的数据和类的定义是否为空。分别循环调用 dumpMethod()函数对直接函数和虚函数进行反编译，类数据头的直接方法的大小作为循环反编译直接方法的循环条件，即 pClassData—>header.directMethodsSize，类数据头的虚方法的大小作为循环反编译直接方法的循环条件 pClassData—>header.virtualMethodsSize。相关代码如代码清单 4.8 所示。

代码清单 4.8　dalvik/dexlist:dumpclass()函数源代码

```
void dumpClass(dexFile * pdexFile, int idx)
{
    const dexClassDef * pClassDef;
    dexClassData * pClassData;
    const u1 * pEncodedData;
    const char * fileName;
    int i;

    pClassDef=dexGetClassDef(pdexFile, idx);
    pEncodedData=dexGetClassData(pdexFile, pClassDef);
    pClassData=dexReadAndVerifyClassData(&pEncodedData, NULL);

    if (pClassData==NULL) {
        fprintf(stderr, "Trouble reading class data\n");
        return;
    }
```

```
    if (pClassDef->sourceFileIdx==0xffffffff) {
        fileName=NULL;
    } else {
        fileName=dexStringById(pdexFile, pClassDef->sourceFileIdx);
    }
/* 在源文件定义每个类的指针,所以可能应该被打印出来。然而需要协调的工具,解析这个输出 */
    for (i=0; i<(int) pClassData->header.directMethodsSize; i++) {
        dumpMethod(pdexFile, fileName, &pClassData->directMethods[i], i);
    }

    for (i=0; i<(int) pClassData->header.virtualMethodsSize; i++) {
        dumpMethod(pdexFile, fileName, &pClassData->virtualMethods[i], i);
    }

    free(pClassData);
}
```

dumpMethod()函数主要完成反编译一个方法的功能。获得 Dex 文件中类的方法的编号和方法的名字。相关代码如代码清单 4.9 所示。

代码清单 4.9 dalvik/dexlist:dumpMethod()函数源代码

```
void dumpMethod(dexFile* pdexFile, const char* fileName,
const dexMethod* pdexMethod, int i)
{
    const dexMethodId* pMethodId;
    const dexCode* pCode;
    const char* classDescriptor;
    const char* methodName;
    int firstLine;

/* 抽象和本地方法没有获得列表 */
    if (pdexMethod->codeOff==0)
        return;

    pMethodId=dexGetMethodId(pdexFile, pdexMethod->methodIdx);
    methodName=dexStringById(pdexFile, pMethodId->nameIdx);

    classDescriptor=dexStringByTypeIdx(pdexFile, pMethodId->classIdx);

    pCode=dexGetCode(pdexFile, pdexMethod);
    assert(pCode !=NULL);

/* 如果文件名是空的,然后将其设置为可打印的,以便它更易于解析 */
/* 一种方法可能会重载其类的默认源文件,指定一个不同的调试信息,这种可能性应该在这里处
理 */
```

```
    if (fileName==NULL || fileName[0]==0) {
        fileName="(none)";
    }

    firstLine=-1;
    dexDecodeDebugInfo(pdexFile, pCode, classDescriptor, pMethodId->protoIdx,
        pdexMethod->accessFlags, positionsCallback, NULL, &firstLine);

    char* className=descriptorToDot(classDescriptor);
    char* desc=dexCopyDescriptorFromMethodId(pdexFile, pMethodId);
    u4 insnsOff=pdexMethod->codeOff + offsetof(dexCode, insns);

    if (gParms.methodToFind != NULL &&
        (strcmp(gParms.classToFind, className) != 0 ||
         strcmp(gParms.methodToFind, methodName) != 0))
    {
        goto skip;
    }

    printf("0x%08x %d %s %s %s %s %d\n",
        insnsOff, pCode->insnsSize * 2,
        className, methodName, desc,
        fileName, firstLine);

skip:
    free(desc);
    free(className);
}
```

Usage()是 Dex 文件加载过程中辅助函数，对 dex 文件进行打印。相关代码如代码清单 4.10 所示。

代码清单 4.10　usage 函数源代码

```
void usage(void)
{
    fprintf(stderr, "Copyright (C) 2007 The Android Open Source Project\n\n");
    fprintf(stderr, "%s: dexfile [dexfile2 ...]\n", gProgName);
    fprintf(stderr, "\n");
}

/* 定位表的回调函数，我们仅仅想要获得在这个方法中第一行的行号，这可能对应着从表中的第一
个入口。 */
static int positionsCallback(void* cnxt, u4 address, u4 lineNum)
{
    int* pFirstLine= (int*) cnxt;
    if (*pFirstLine==-1)
```

```
            * pFirstLine= lineNum;
        return 0;
    }
```

通过对上述 dexlist 工具的使用函数的分析，结合 Dex 文件的结构对 dexlist 工具对类的操作更加清晰。

4.5 dexopt 工具

4.5.1 dexopt 工具简介

dexopt 工具作为一个 Dex 文件优化的工具，主要完成由 Dex 或者 zip 文件生成 Odex 文件，此过程实现对 Dex 文件的优化，提高 Dex 文件载入和类加载的效率，源码文件目录：Android_dalvik_source\dexopt\OptMain.cpp 命令行的优化和验证入口点，这个是文件编译生成 Dex 优化工具。

4.5.2 dexopt 工具使用方法

在 dexopt 工具源码中提到了如下三种启动 dexopt 工具的方法。

(1) 从 VM 启动。这需要十几个参数，其中之一是一个文件的描述符，该描述符作为输入和输出。这使我们可以不用考虑 Dex 数据的来源。

(2) 从已安装的或其他本机应用程序启动。通过一个用于 zip 文件的文件描述符，一个用于输出的文件描述符，一个用于调试消息的文件名进行传递。对于正在进行的活动可能有多种假设(允许验证+优化，引导类的路径是在 BOOTCLASSPATH 等)。

(3) 在主机上预先优化的构建过程中启动。与(2)类似，只不过它采用的是文件描述符，而不是文件名。

Android 系统 4.0.4 版本源码的 build/tools/dexpreopt/dexopt-wrapper/目录下的 dexopt-wrapper 主程序源码中的 dexOptwripper.cpp 文件实现过程中调用的是/system/bin/dexopt 程序，也就是本节介绍的 dexopt 工具。dexopt 工具并没有提供直接调用的命令，因此，dexopt-wrapper 工具可以帮助用户通过命令调用 dexopt 工具，完成相关的优化操作。具体的操作步骤如下。

第一步，启动模拟器

由于运行工具或是模拟器之前，都需要设置环境变量，前文已多次使用并给出了详细方法，后文不再对这一步多做说明。

第二步，运行 dexopt-wrapper 工具完成优化

首先将 dewopt-wrapper 工具放入指定位置，将目录转换到 dexopt-wrapper 工具所在目录，命令为 Adb push dewopt-wrapper /data/local/。然后给 dewopt-wrapper 工具赋予权限，命令为 adb shell chmod 777 /data/local/dewopt-wrapper，使用相同的 push 方法将待优化的文件转换目录，命令为 adb push Fibonacci.apk /data/local，紧接着调用命令./dexopt-wrapper Fibonacci.apk testopt.odex，其中 testopt.odex 为生成的优化文件名字。完成优化后，使用 pull 命令将 odex 文件拖曳出来，命令为：adb pull /data/local/testopt.odex。具

体实现流程如图 4.9 所示。

```
cheng@cheng-laptop:~/dexopt$ adb push dexopt-wrapper /data/local/
71 KB/s (5512 bytes in 0.075s)
cheng@cheng-laptop:~/dexopt$ adb shell chmod 777 /data/local/dexopt-wrapper
cheng@cheng-laptop:~/dexopt$ adb push Fibonacci.apk /data/local/
1805 KB/s (233354 bytes in 0.126s)
cheng@cheng-laptop:~/dexopt$ ./dexopt-wrapper Fibonacci.apk testopt.odex
```

图 4.9　Dex 文件优化命令示意

Dex 文件的优化始于 Android 源码中 frameworks 层，一些参数通过调用命令的方式无法直接获得，在 dexopt-wrapper 工具运行过程中，实际是为 dexopt 工具中 Dex 文件优化提供这些参数，在优化过程中，将 dexopt 的路径字符串以宏定义的形式传入，对文件后缀进行匹配判断是否为 zip 文件，准备 zip 文件描述符，生成 Odex 文件的文件描述符，输出文件的名字，Dex 文件优化选项。以上这些是 dexopt 优化过程中的核心参数，用于完成 Dex 文件优化。

dexopt 优化工具主要完成对 Dex 文件的优化操作。对 Dex 文件进行解析，提高 Dex 文件的载入速度并提高类加载的效率。类加载相关分析将在本丛书第二卷类加载章中介绍。

4.6　dvz 工具

4.6.1　dvz 工具简介

dvz 工具作为一个 Framework 框架调试的工具，作用是从 Zygote 进程中触发出一个新的进程，这个进程也是一个 Dalvik 虚拟机。该进程与 DalvikVM 启动的虚拟机相比，区别在于该进程中已经预装了 Framework 的大部分类和资源。工具的相关源码位于 Android 虚拟机根目录下，即 Android_dalvik_source\dvz。源码内容比较少，有兴趣的读者可以阅读一下。

4.6.2　dvz 工具使用方法

首先在 Eclipse 环境下编写一个简单的 Android 应用程序，并将这个 Android 应用程序打包成 apk 文件，相信一个 Android 程序开发人员对于打包成 apk 文件的过程并不陌生。相关代码如代码清单 4.11 所示。

代码清单 4.11　手动编写 dvz 运行源代码

```
package com.Android.dvztest;
import java.io.IOException;
import Android.os.Bundle;
import Android.app.Activity;
import Android.view.Menu;
import Android.os.Debug;

public class MainActivity extends Activity {
```

```
        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);
            System.out.println("Hello");

        }
        public static void main(String[] args)
        {
             System.out.println("Hello dalvik");
        }
    }
```

只是在初始程序的基础上加入 static main()函数,使用 adb push 将 apk 程序放入模拟器中。调用 dvz -classpath 包名称类名。虽然通过这种方式完成了一个类的启动,但 MainActivity 类并不是该应用程序的入口类,一个 apk 的入口类是 ActivityThread 类,Activity 类仅仅是被回调的类,因此不可以通过 Activity 类来启动一个 apk,dvz 工具仅用于 Framework 开发过程的调试。

小结

本章相对来说具有很强的操作性,从 Dalvik 虚拟机的工具介绍,我们看到了 Dalvik 虚拟机运行过程使用的系统工具,同时从更多方面了解 Dalvik 虚拟机,通过工具的使用对虚拟机内部的实现机制更加清晰,如何实现 Dex 文件优化,如何对封装的 apk 进行反编译,如何对 Android 程序源码进行调试分析内存泄漏,如何对 Android 程序运行过程中生成的 trace 文件进行分析。对于大多数从事 Android 程序开发的程序员来说,熟练运用这些工具,分析 Android 程序源码将变得得心应手,内存泄漏作为常常被程序员在编码过程中忽略的问题,资源使用后未释放,有时候只有当程序大量消耗内存卡死的时候才体现出来,通过对堆栈和程序运行过程中 trace 文件进行分析,可以准确定位程序中源码语句,有效地提高程序运行效率。

第 5 章 开发分析工具

本章主要内容

✍如何对 Android 程序进行分析？
✍通过开发分析工具对 Android 源码的调试方法有哪些？
✍如何通过 trace 文件跟踪分析代码？
✍如何分析堆栈信息准确定位程序运行的内存泄漏？
✍DDMS 工具是怎样完成程序源码分析的？

5.1 本章概述

在 Dalvik 中的工具很多，在第 4 章对 Dex 文件分析工具介绍的基础上，本章结合 Dalvik 的开发分析工具，围绕源码分析的相关操作展开，使用 Dalvik 中的开发分析工具对 Android 程序源码进行分析，使用 DDMS 完成对 Android 程序源码进行调试和堆栈分析的功能，可以有效地提高程序运行的效率和安全性。

本章主要介绍程序开发过程中分析调试工具，提高程序的安全性和效率。为了解决在程序开发过程中困扰程序员的内存泄漏、堆栈溢出以及一些不易发现的问题，Dalvik 虚拟机也提供了对 Android 程序源码调试的工具 Heap Profile，分析程序的内存、堆栈耗时以及调用信息，完成代码的优化功能。分析程序运行过程中的 trace 的分析工具，对程序执行的每步进行分析。DDMS 工具完成对开发程序的具体包进行调试分析，通过对包中的进程、线程以及开发包含的具体变量的内存（堆、栈）的申请分配情况的分析调试有助于对开发应用程序内存分配情况进行优化，开发高效的程序。

5.2 trace 文件分析工具

5.2.1 trace 文件分析工具简介

trace 文件分析工具是为了更加方便地了解 Android 程序中的堆栈调用的次序、消耗时间、调用次数等信息以及 Android 程序在执行过程中的调用关系和运行机制，用于分析自己编写的 Android 程序的整个实现流程以及相关函数的执行顺序，实现对局部代码段的分析，有效提高对代码的分析效率。

由于 Android 程序开发大都是在 Windows 操作系统下进行的，所以，在安装了 Android 开发环境的操作系统下生成 trace 文件，对于该工具生成的 trace 文件，需要使用 traceview 工

具对其进行分析。

5.2.2 trace 文件分析工具使用方法

第一步,在代码段中加入调试命令

在需要进行调试的代码前一行加入 Android. os. Debug. startMethodTracing ("/sdcard/test");,表示调用 Android 调试模式下的调试方法,括号内表示生成文件的对应路径和生成文件的文件名,示例中 stratMethodTracing()方法,参数为 SD 卡中文件名为 test 的 trace 文件。在结束调试的代码的后一行加入 Android. os. Debug. stopMethodTracing();调试生成 trace 文件使用到的代码如代码清单 5.1 所示。

代码清单 5.1 手动编写 trace 调试程序源代码

```
package com.yucheng.fibonacci;
import java.io.IOException;
import Android.os.Bundle;
import Android.app.Activity;
import Android.view.Menu;
import Android.os.Debug;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        test();

    }
    public static void test()
    {
         Android.os.Debug.startMethodTracing("/sdcard/test");
         System.out.println("Hello dalvik");
         Android.os.Debug.stopMethodTracing();
    }
}
```

在上述程序中再自定义一个 test()方法,在这个方法里加入了上述提到的 trace 的调试代码,用于生成 trace 文件。

由于在运行过程中 apk 程序需要对 Android 模拟器中的 SD 卡进行写入操作,所以需要在 Android 程序工程的 AndroidManifest. xml 文件中加入权限允许的代码,否则会有 permission denied 的错误提示。相关代码如下:

```
<uses-permission Android:name="Android.permission.WRITE_EXTERNAL_STORAGE">
</uses-permission>
```

```
<uses-permission Android:name="Android.permission.MOUNT_UNMOUNT_FILESYSTEMS">
</uses-permission>
```

如果不在 AndroidManifest.xml 的源文件中添加，也可以在如图 5.1 所示的方法中进行添加。

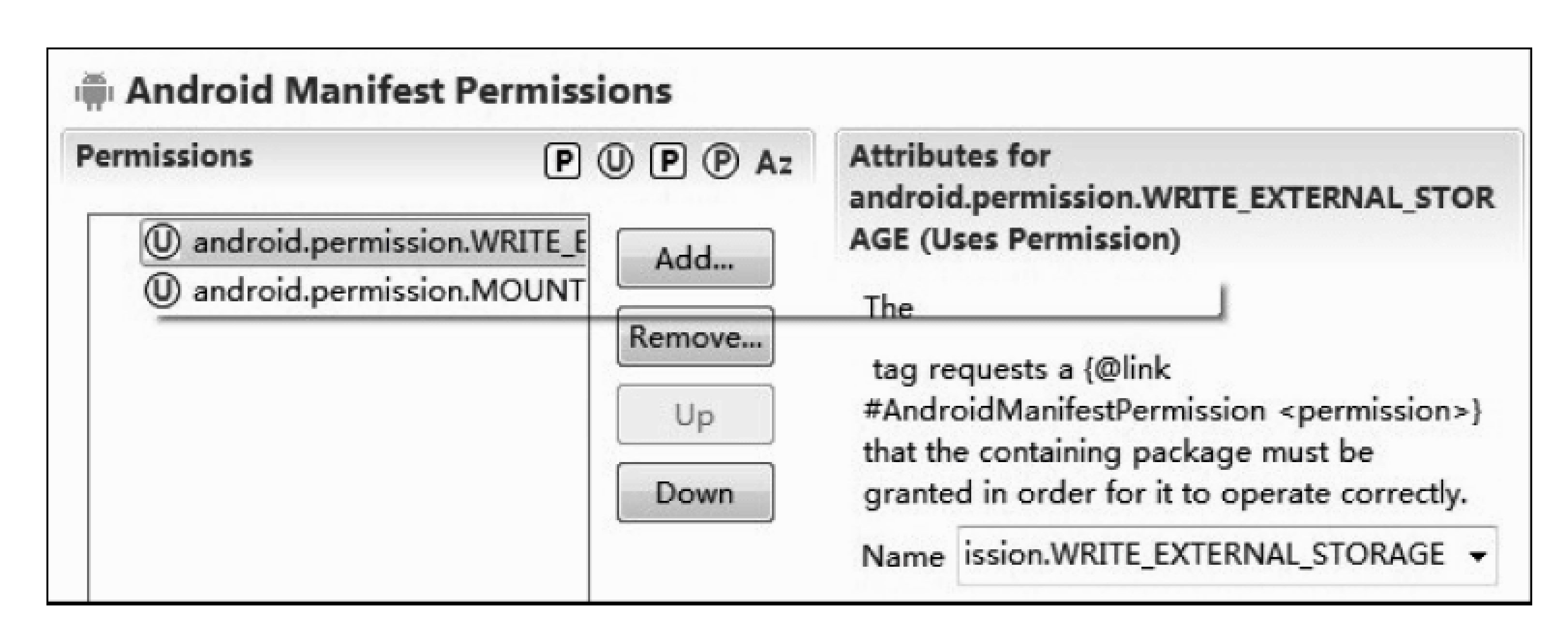


图 5.1　添加 Android 应用程序的权限

第二步，发布 apk 程序，生成格式为 trace 的文件

为了保证程序的运行流畅和生成的 trace 文件可以成功保存，建议在创建 Android 模拟器时，将 RAM 设置为 512MB，将 SD 卡容量设置为 512MB 或者更大，在 Eclipse 平台下对编写的 Android 程序进行编译生成对应的 apk 文件，将生成的 apk 文件传入 Android 模拟器的 SD 卡中。

选择 debug as 命令在 debug 模式下运行 Android 程序。运行过程中相关日志如图 5.2 所示，在图中第一行可以清晰看到 trace 文件生成。

```
07-18 07:32:20.095   884   884   com.yucheng.fibon...   dalvikvm          TRACE STARTED: '/sdcard/test.trace' 8192KB
07-18 07:32:20.515   884   884   com.yucheng.fibon...   dalvikvm          +++ active profiler count now 1
07-18 07:32:20.527   884   884   com.yucheng.fibon...   System.out        Hello dalvik
07-18 07:32:20.527   884   884   com.yucheng.fibon...   dalvikvm          +++ active profiler count now 0
07-18 07:32:20.885   884   884   com.yucheng.fibon...   dalvikvm          TRACE STOPPED: writing 45 records
07-18 07:32:22.816   884   884   com.yucheng.fibon...   gralloc_gold...   Emulator without GPU emulation detected.
```

图 5.2　调试生成 trace 文件日志信息

第三步，使用 traceview 工具分析 trace 文件

将生成的 trace 格式文件从 SD 卡中复制到 PC 端。使用 traceview 软件对生成的 trace 文件进行分析。使用 adb pull 命令将 SD 卡中的文件复制到 PC 端，这个命令已在前面章节多次使用，具体命令为 adb pull /sdcard/test.trace D:/，实例命令表示将 sdcard 目录下的 test.trace 文件复制到 D 盘根目录下，操作过程如图 5.3 所示。

```
C:\Users\yu>adb pull /sdcard/test.trace D:/
17 KB/s (2884 bytes in 0.157s)
```

图 5.3　将 SD 卡上的文件复制到 PC 端

使用 traceview 工具打开 test.trace 文件，具体操作命令如下：traceview D:\test.trace。其中调用 traceview 为使用 traceview 工具，后面加入待打开文件路径，操作的过程如图 5.4 所示。

```
C:\Users\yu>traceview D:\test.trace
The standalone version of traceview is deprecated.
Please use Android Device Monitor (tools/monitor) instead.
```

图 5.4　将 SD 卡上的文件复制到 PC 端

使用 traceview 工具打开刚刚使用 debug 调试模式生成的 trace 文件，如图 5.5 所示。

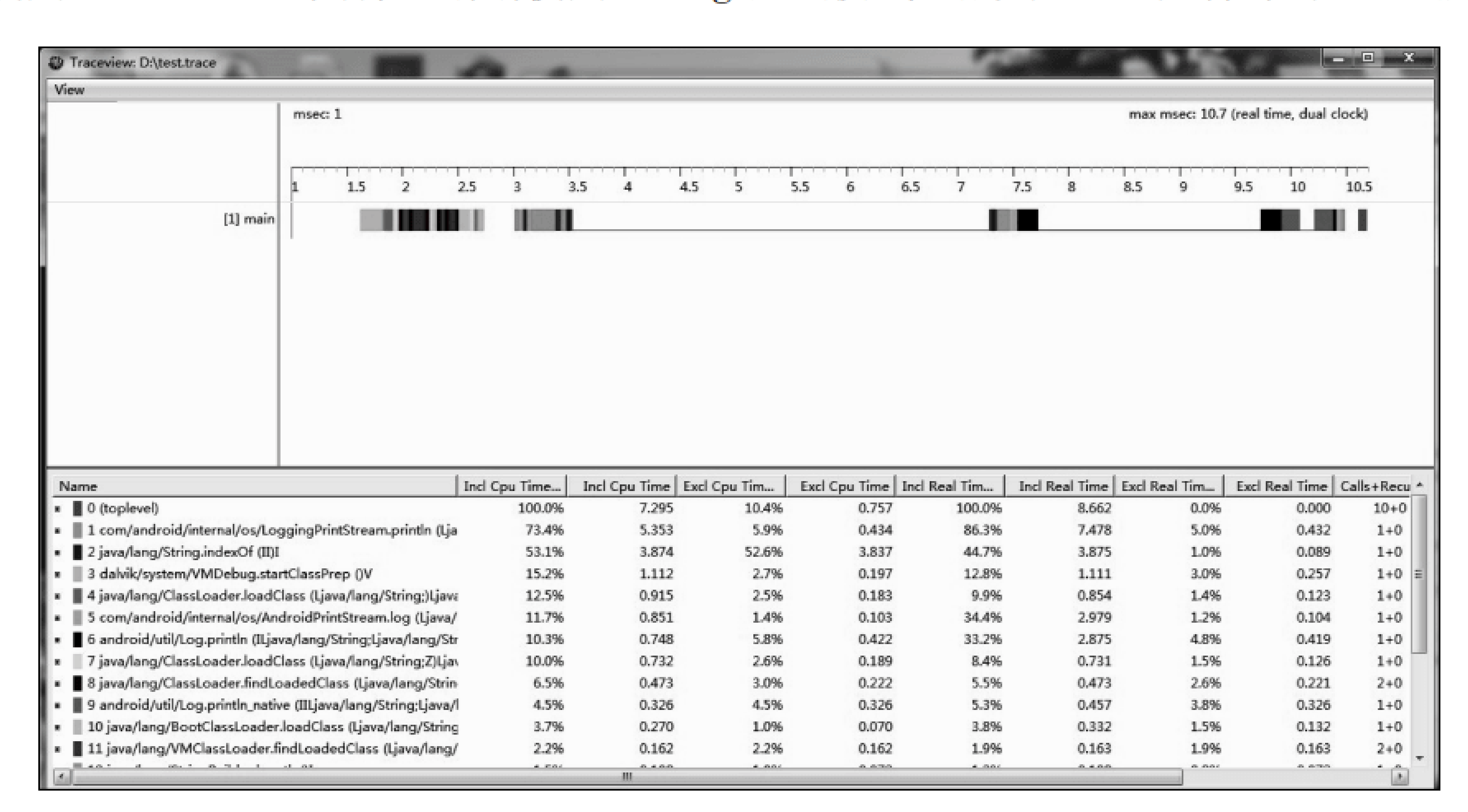


图 5.5　打开 trace 文件示意图

分析 trace 文件的过程中，可以将鼠标放在窗口上方的水平轴上，可以清晰地看到程序中每个线程调用方法的启动和停止时间。分析程序每一步的流程，定位程序源码中存在问题的代码。

5.3　Heap Profile 工具

5.3.1　Heap Profile 工具简介

说到程序的内存泄漏大家并不陌生，平时编写的程序中常常会出现内存泄漏，但是往往找到内存泄漏却是棘手的，Heap Profile（以下简称 Hprof）是一个调试程序代码并分析 Android 程序内存情况的工具。通过在程序运行过程中，在指定目录下生成 hprof 格式的文件，使用 hprof-conv 工具对 hprof 格式文件进行转换，结合内存分析工具 Memory Analyzer Tool（以下简称 MAT）对 hprof 标准格式文件分析，了解程序运行过程中的内存情况，发现程序代码中存在的潜在问题，便于程序员对代码进行修改优化，以免存在问题，避免产生程序大量占用内存的情况，甚至产生被利用的漏洞。

由于进行 Android 程序开发的程序员主要是在 Windows 操作系统下完成，Hprof 是对源码进行调试的工具，为了程序编译运行的方便，Hprof 工具的使用也主要在 Windows 操作系统下进行。

分析 Android 操作系统程序的方法有两种，一种是利用 Android 平台提供的 Android.util.Log 通过 log 信息来分析错误发生的原因；另外一种是通过设置断点，一步一步地跟踪

程序发现问题。这两个方法非常有效,介绍相关方法的资料也很多。

还有一类常见的问题就是 Memory Leak。对内存泄漏这类问题,以上两种方法不是很有效,在 DDMS 工具里面,也基本上只能查看到 Heap 的使用情况,对分析问题帮助不大。可以利用 Eclipse MAT 工具来分析此类问题。本节将通过 Memory Analyzer 这一个快速并且功能强大的 Java heap 分析器分析 hprof 文件,能够帮助查找内存泄漏并减少内存消耗。

5.3.2　Heap Profile 工具使用方法

第一步,在程序运行过程中生成 hprof 文件。

在程序代码中加入生成 hprof 文件所需的调试语句,在引入的头文件中包括:

```
import java.io.IOException;
import Android.os.Debug;
```

使用 dumpHprofData 函数将生成的 hprof 文件导出到模拟器中事先建立的 tmp 文件夹下,核心的调试代码段如下:

```
try {
    Android.os.Debug.dumpHprofData("/data/tmp/input.hprof");
}
catch (IOException ioe)
{
System.out.println("a error");
    }
```

作者使用的简单示例的相关代码如代码清单 5.2 所示。

代码清单 5.2　手动编写堆栈分析源代码

```
package com.yucheng.fibonacci;
import java.io.IOException;
import Android.os.Bundle;
import Android.app.Activity;
import Android.view.Menu;
import Android.os.Debug;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        long result= fib_java(3);
        System.out.println(result);
        try {
```

```
                Android.os.Debug.dumpHprofData("/data/tmp/input.hprof");
            } catch (IOException ioe) {
                System.out.println("a error");
            }

    }

    public long fib_java(int num)
    {
        if (num==0){
            return num;
        }
        else
        {
            return 0;
        }
    }
}
```

第二步,将 hprof 文件从模拟器中复制到 PC 端,使用 hprof-conv 工具将 hprof 文件转换成标准格式。

将目录转换到 hprof 文件所在的文件路径,使用 adb pull 命令将生成的 hprof 文件从存储路径复制到 PC 端。但是这时候生成的 hprof 文件并不是标准格式的,需要使用 Android SDK 中的 hprof-conv 工具进行转换,首先将文件路径转换到 hprof,然后运行进行标准格式的转换相关的命令为:hprof-conv input. hprof out. hprof。具体实现流程如图 5.6 所示。



图 5.6 将生成的 hprof 格式文件转换成标准格式

如果直接使用 MAT 进行分析,将提示"Unknown HPROF Version (JAVA PROFILE 1.0.3) (java. io. IOException)"异常,如图 5.7 所示。

点拨 不要以为是 MAT 的版本不对,其实是 Android 的 hprof 文件在这里需要进行转换成标准格式才可以使用 MAT 打开,都是相同的文件格式,不知道谷歌在这里做了什么文章,难道是做了什么优化?有兴趣的读者可以研究下。

第三步,使用 MAT 分析。

在生成标准 hprof 文件格式后,需要使用内存分析工具对 hprof 文件进行分析。作者在前段时间做的一个 Android 项目的过程中,出现 Exception in thread "main" java. lang.



图 5.7　未转换的非标准格式 hprof 文件提示错误

OutOfMemoryError：Java heap space 这个错误，所以需要查找原因，就用到 MAT。由于 MAT 不是 Eclipse 自带的工具所以需要在 Eclipse 下手动配置 MAT。MAT 的配置过程如下。

（1）下载 Eclipse MAT 工具。

从 http://www.eclipse.org/mat/downloads.php 网站上下载 Eclipse MAT。

下载页如图 5.8 所示。

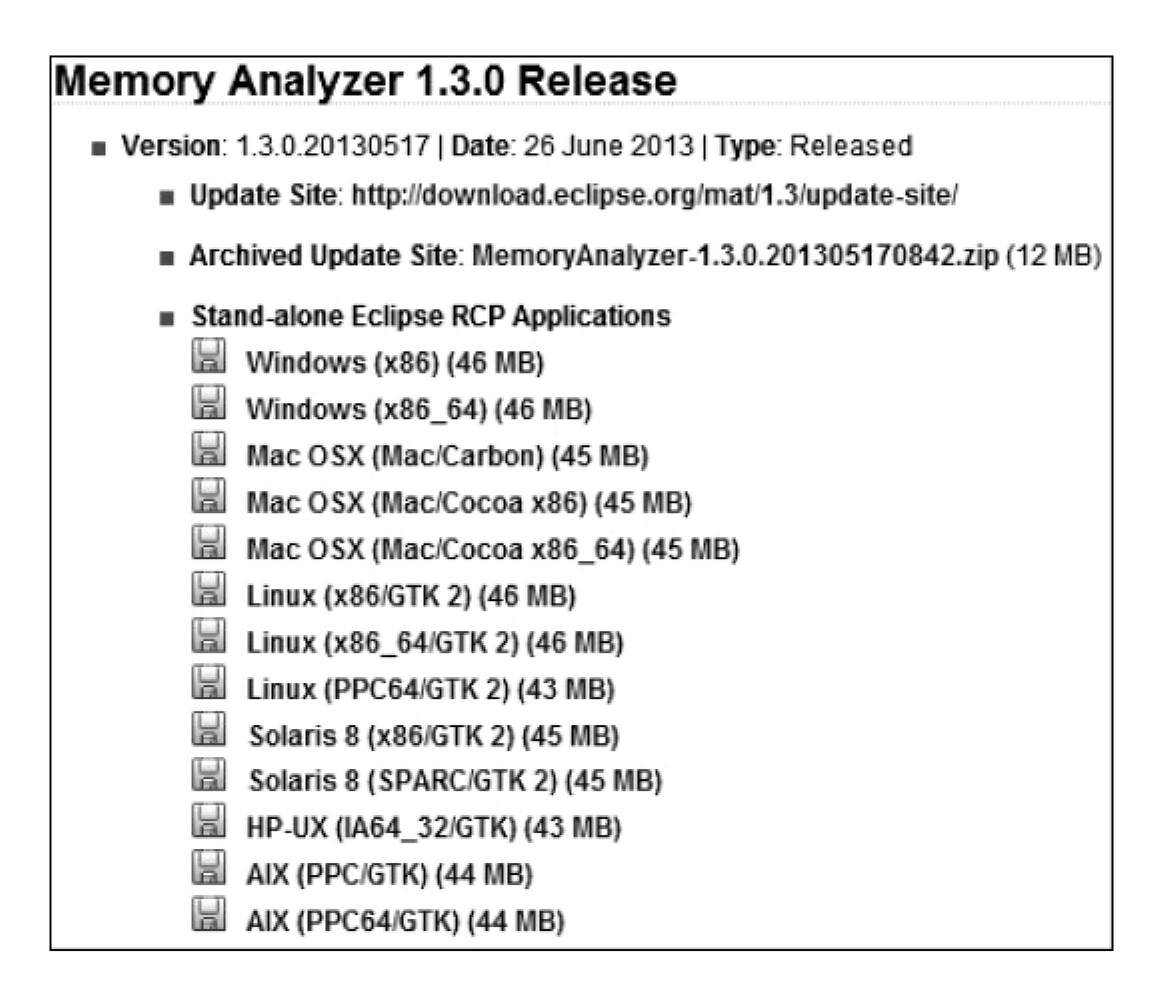


图 5.8　下载 MAT 的页面

（2）下载之后将压缩包解压，作者使用的继承了 Android SDK 的 Eclipse 开发环境，放置到 Eclipse 的\adt-bundle-windows-x86-20130522\eclipse\dropins 目录下，如图 5.9 所示。

返回到上一级目录，即\eclipse\dropins 目录下，新建一个文件，mat.link 文件内容如下：path=MAT 的解压路径。本例中的 mat.link 文件的内容如下，注意本例使用“\\”因为是转义字符。

130522 ▸ eclipse ▸ dropins ▸ MemoryAnalyzer-1.2.1.201211051250 ▸

新建文件夹

名称	修改日期	类型	大小
features	2012/11/6 3:16	文件夹	
plugins	2012/11/6 3:15	文件夹	
web	2012/11/6 3:13	文件夹	
artifacts.jar	2012/11/6 3:15	Executable Jar File	2 KB
content.jar	2012/11/6 3:15	Executable Jar File	13 KB
index.html	2012/11/6 3:13	HTML 文档	2 KB
notice.html	2012/11/6 3:13	HTML 文档	10 KB
site.xml	2012/11/6 3:15	XML 文档	2 KB

图 5.9 解压 MAT 到 Eclipse 安装目录

```
path=D:\\adt-bundle-windows-x86-20130522\\adt-bundle-windows-x86-20130522\\
eclipse\\dropins\\MemoryAnalyzer-1.2.1.201211051250
```

(3) 重新启动 Eclipse,使用 MAT 进行分析。

在成功安装了 MAT 后,单击菜单栏上面的 Windows 下的 Preference 选项,会在弹出的窗口中显示出 Memory Analyzer 工具。再从菜单栏中选择"文件"→"打开"选项,打开要分析的 hprof 文件。打开 hprof 文件的主界面如图 5.10 所示。

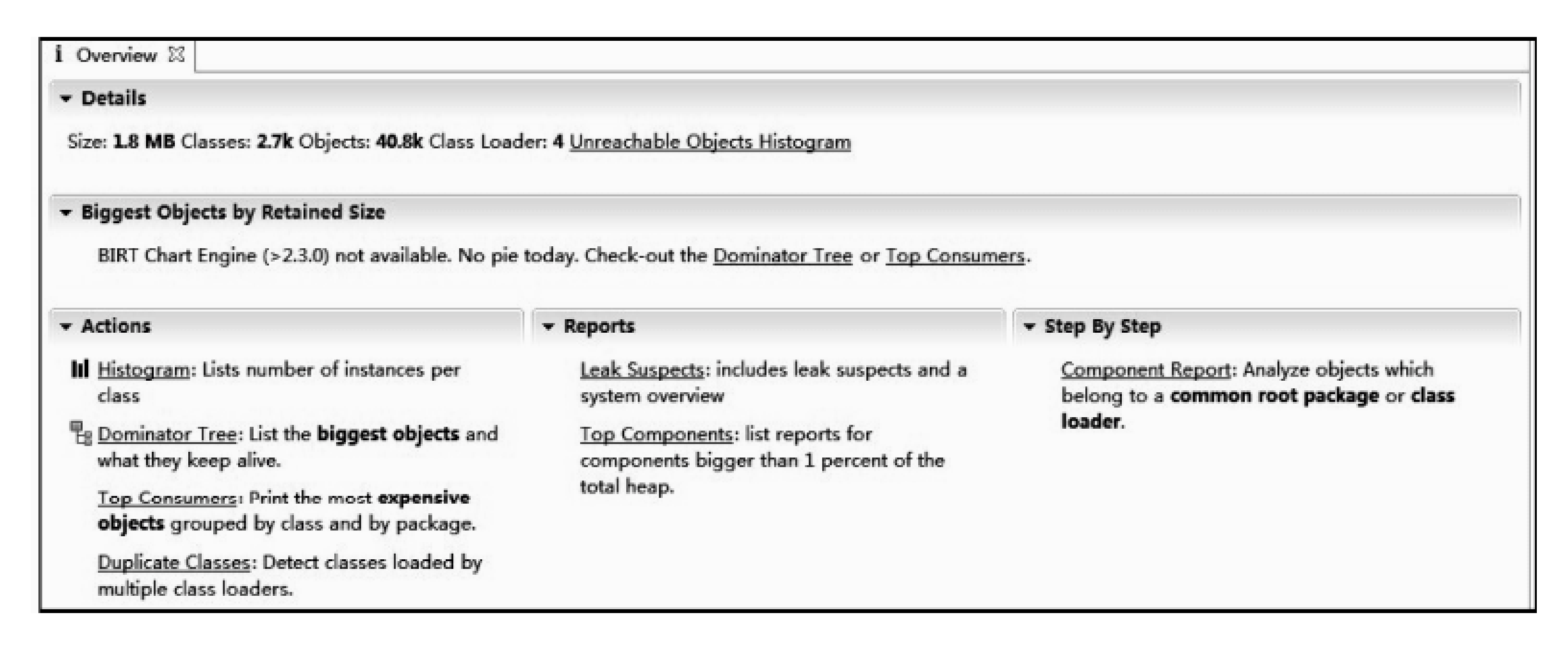


图 5.10 生成的内存分析报告

单击图 5.10 中的 Reports→Leak Suspects 则可以进一步看到更详细的内存泄漏疑点。相关的内存泄露疑点的信息如图 5.11 所示。

想要查看更多更全的信息,可以通过 table of content 获得,具体信息如图 5.12 所示。

MAT 是一个很强大的内存分析工具,对于分析程序源码较大的 Android 工程帮助很大,可以帮助快速定位产生内存泄漏的类。在平时编写程序的过程中,内存泄漏往往是容易被忽略的,但是当一个小的问题扩大化的时候,就会产生一个巨大的效果。对于 PC 端的程序,当内存泄漏比较严重的时候,会出现大量占用内存使内存计算机卡机的情况,对于 Android 应用程序,由于本身移动设备的内存容量就有限,内存泄漏现象将导致移动设备卡机,甚至导致机器重启。

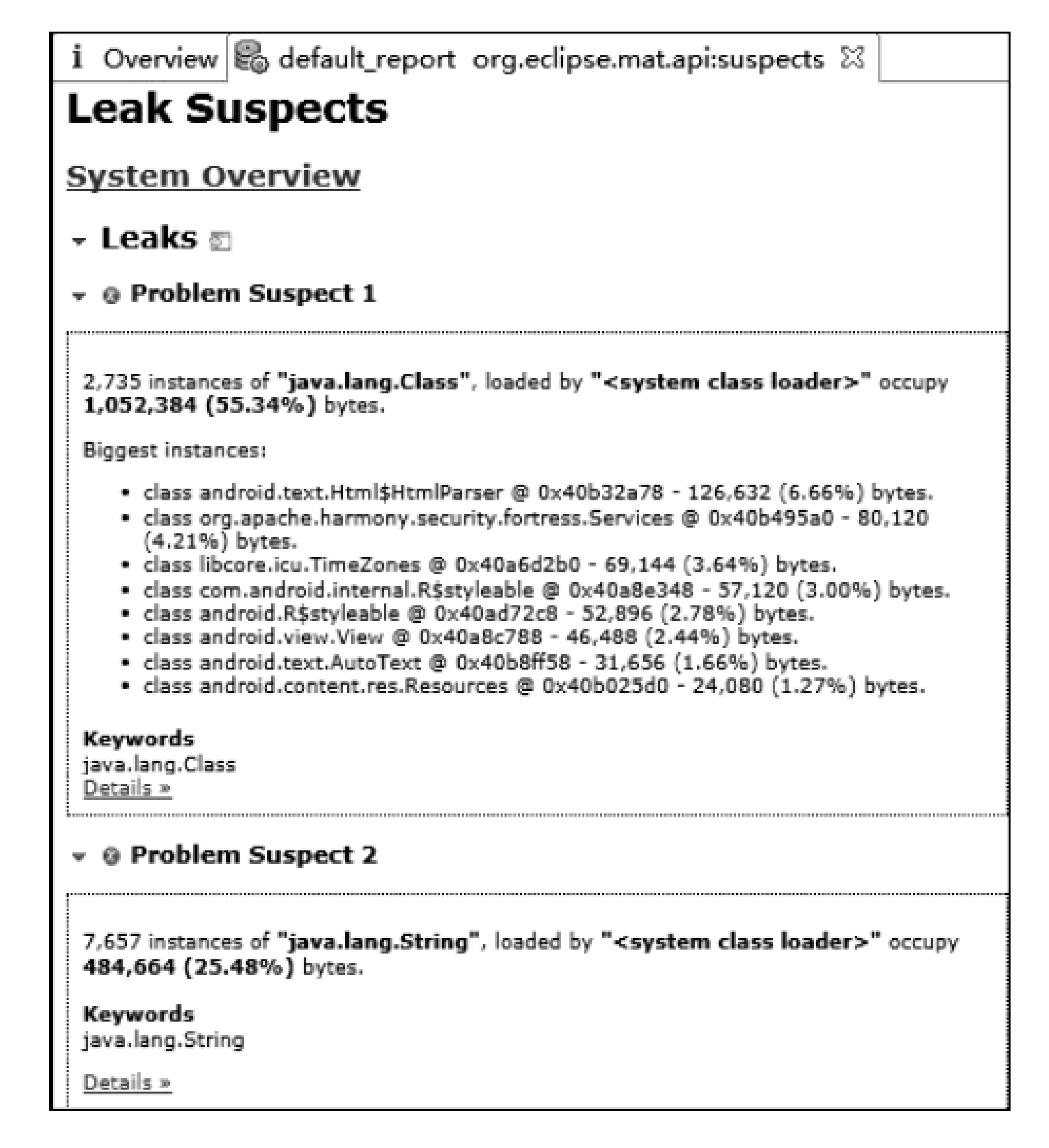


图 5.11　内存泄漏疑点

Table Of Contents

- System Overview
  - Heap Dump Overview
  - System Properties
  - Thread Overview
  - Top Consumers
    - Biggest Objects (Overview)
    - Biggest Objects
    - Biggest Top-Level Dominator Classes (Overview)
    - Biggest Top-Level Dominator Classes
    - Biggest Top-Level Dominator Class Loaders (Overview)
    - Biggest Top-Level Dominator Class Loaders
    - Biggest Top-Level Dominator Packages
  - Class Histogram
- Leaks
  - Overview
  - Problem Suspect 1
    - Description
    - Reference Pattern
  - Problem Suspect 2
    - Description
    - Reference Pattern

图 5.12　内存信息的详细列表

5.4　DDMS 工具

Dalvik 调试监视服务(Dalvik Debug Monitor Service，DDMS)是由 Android 软件开发包提供的调试工具。开发人员可以使用 DDMS 提供的窗口来监视模拟器或真实设备的调试，包括为测试设备截屏，针对特定的进程查看正在运行的线程以及堆信息、Logcat、广播状态信息、模拟电话呼叫、接收 SMS、虚拟地理坐标等。它是几个工具的融合：任务管理器(Task manager)、文件浏览器(File Explorer)、模拟器控制台(Emulator console)和日志控

制台(Logging console)。

5.4.1 启动 DDMS

启动 DDMS 有以下两种方法。

(1) 如果没有使用集成开发环境 Eclipse,那么 DDMS 可以单独的进程运行,位于 Android 软件开发包中/Tools 目录下,双击运行 ddms.bat 文件,如图 5.13 所示。

```
C:\Windows\system32\cmd.exe
The standalone version of DDMS is deprecated.
Please use Android Device Monitor (monitor.bat) instead.
08:42:18 E/ddmlib: 您的主机中的软件中止了一个已建立的连接。
java.io.IOException: 您的主机中的软件中止了一个已建立的连接。
        at sun.nio.ch.SocketDispatcher.write0(Native Method)
        at sun.nio.ch.SocketDispatcher.write(SocketDispatcher.java:33)
        at sun.nio.ch.IOUtil.writeFromNativeBuffer(IOUtil.java:104)
        at sun.nio.ch.IOUtil.write(IOUtil.java:75)
        at sun.nio.ch.SocketChannelImpl.write(SocketChannelImpl.java:334)
        at com.android.ddmlib.JdwpPacket.writeAndConsume(JdwpPacket.java:213)
        at com.android.ddmlib.Client.sendAndConsume(Client.java:642)
        at com.android.ddmlib.HandleHeap.sendREAQ(HandleHeap.java:348)
        at com.android.ddmlib.Client.requestAllocationStatus(Client.java:488)
        at com.android.ddmlib.DeviceMonitor.createClient(DeviceMonitor.java:835)

        at com.android.ddmlib.DeviceMonitor.openClient(DeviceMonitor.java:803)
        at com.android.ddmlib.DeviceMonitor.processIncomingJdwpData(DeviceMonito
r.java:763)
        at com.android.ddmlib.DeviceMonitor.deviceClientMonitorLoop(DeviceMonito
r.java:652)
        at com.android.ddmlib.DeviceMonitor.access$100(DeviceMonitor.java:44)
        at com.android.ddmlib.DeviceMonitor$3.run(DeviceMonitor.java:580)
```

图 5.13 DDMS 通过 dat 工具单独启动

这样 DDMS 将运行在自己的进程中,并自动调用 Android Device Monitor 自动连接到 Emulator。监视器连接界面如图 5.14 所示。

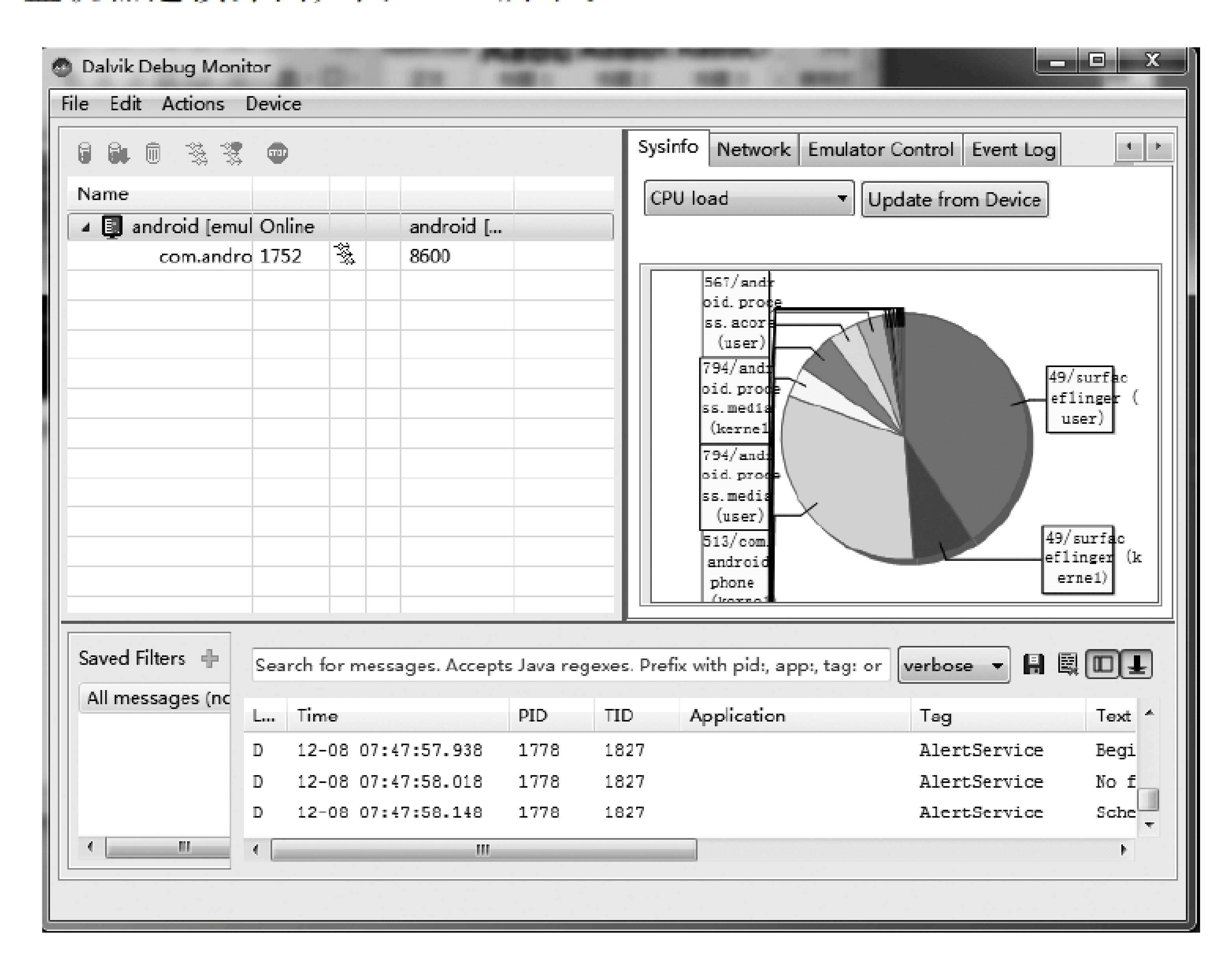


图 5.14 DDMS 监视器连接界面

（2）如果使用集成开发环境 Eclipse，并且安装了 Android 开发工具插件，DDMS 工具就已经结合到了集成开发环境中。通过 DDMS，可以浏览计算机上运行的模拟器实例，并且可以通过 USB 连接真实的 Android 设备进行调试。在 Eclipse 调试程序的过程中启动 DDMS，在 Eclipse 中的界面如图 5.15 所示。

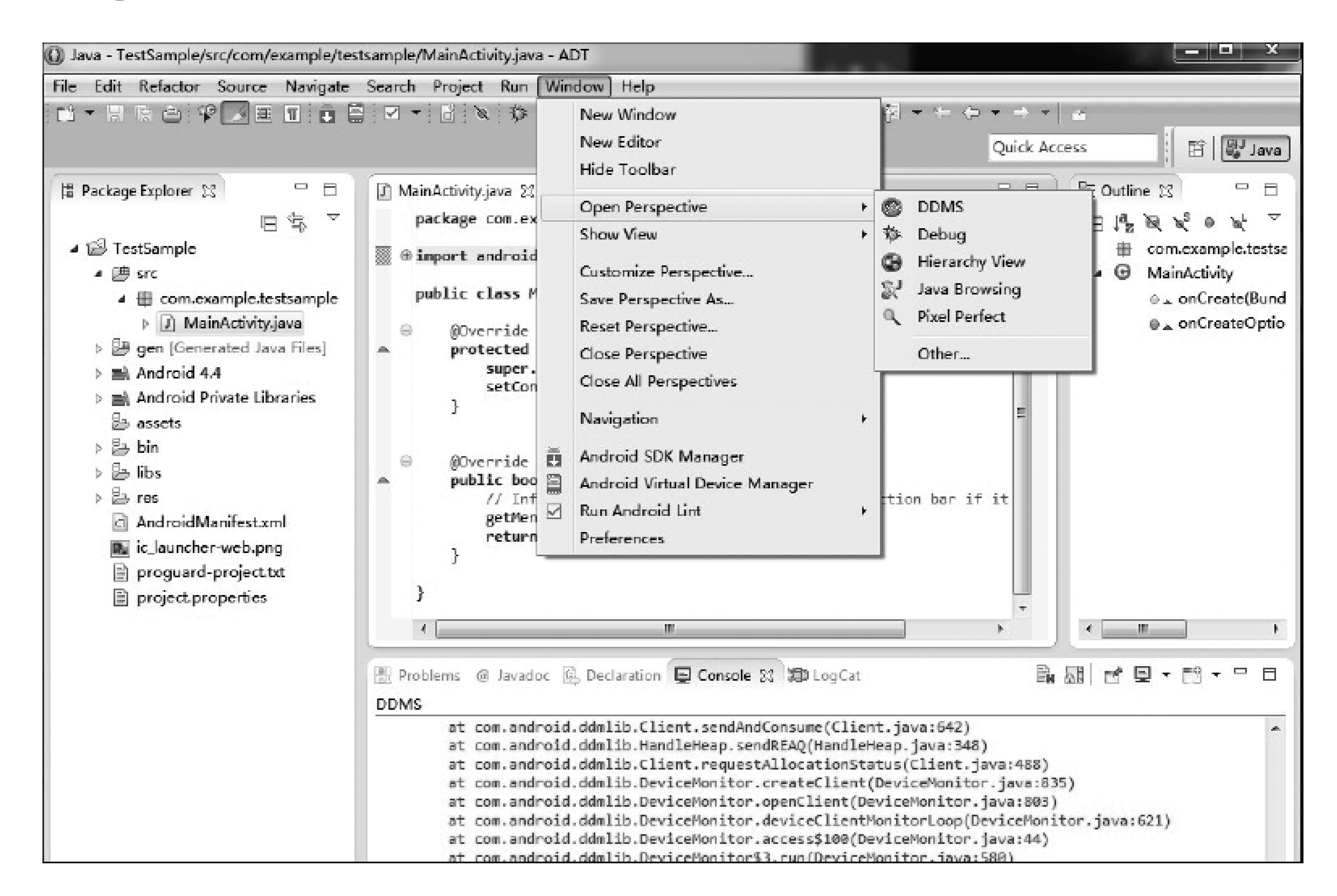


图 5.15　Eclipse 集成 DDMS 启动过程

可以通过 Eclipse 看到 DDMS 工具中包含一个模拟器实例，如图 5.16 所示。

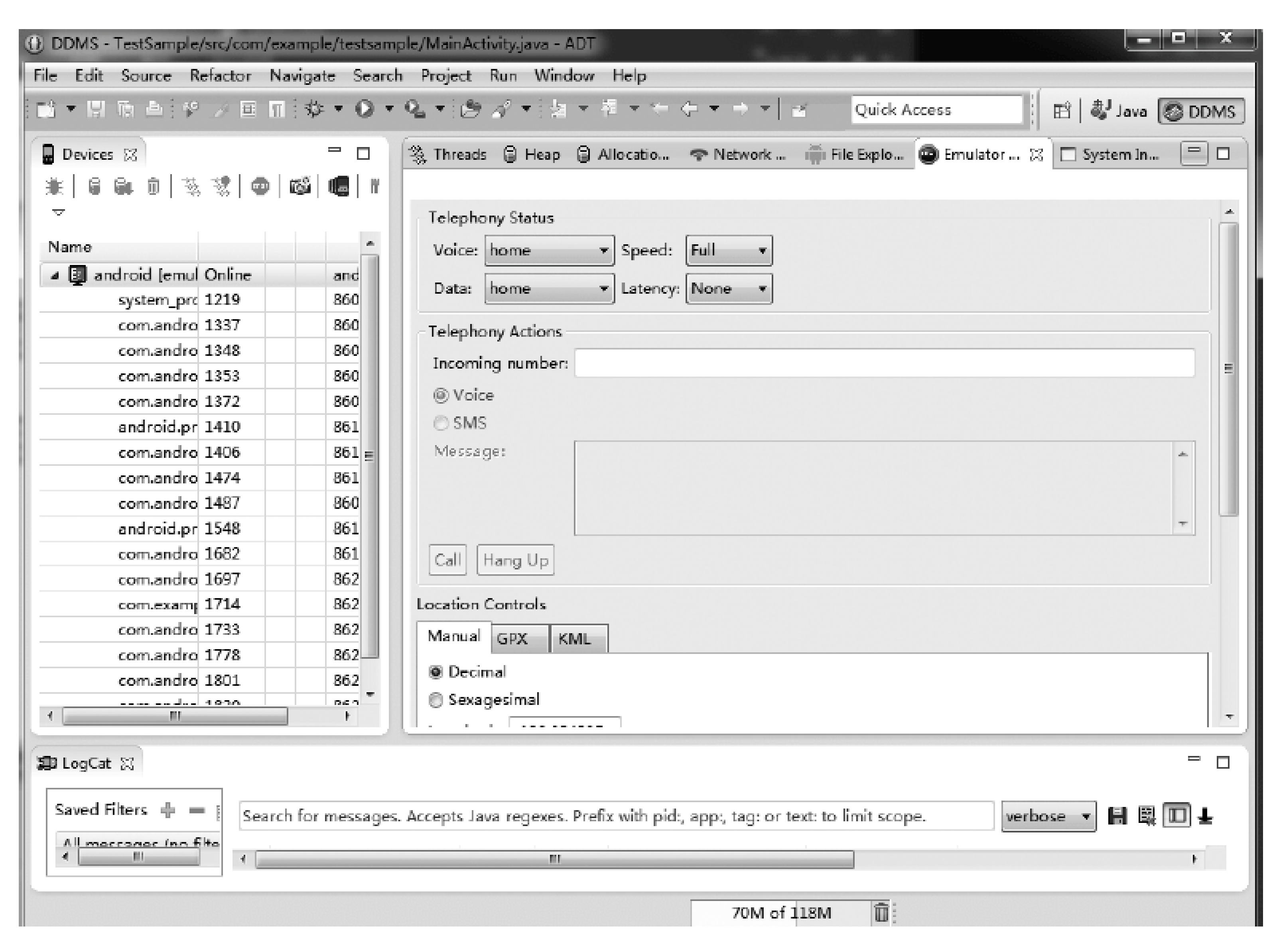


图 5.16　DDMS 调试界面包含的设备

DDMS 对 Emulator 和外接测试机有同等效用。如果系统检测到它们（VM）同时运行，那么 DDMS 将会默认指向 Emulator。并且在同一时间只运行一个 DDMS 实例。其他运行

的 DDMS 会被忽略。

5.4.2 DDMS 原理和特性

DDMS 将搭建起 IDE 与测试终端(Emulator 或者 connected device)的连接,它们应用各自独立的端口监听调试器的信息,DDMS 可以实时监测到测试终端的连接情况。当有新的测试终端连接后,DDMS 将捕捉到终端的 ID,并通过 adb 建立调试器,从而实现发送指令到测试终端的目的。

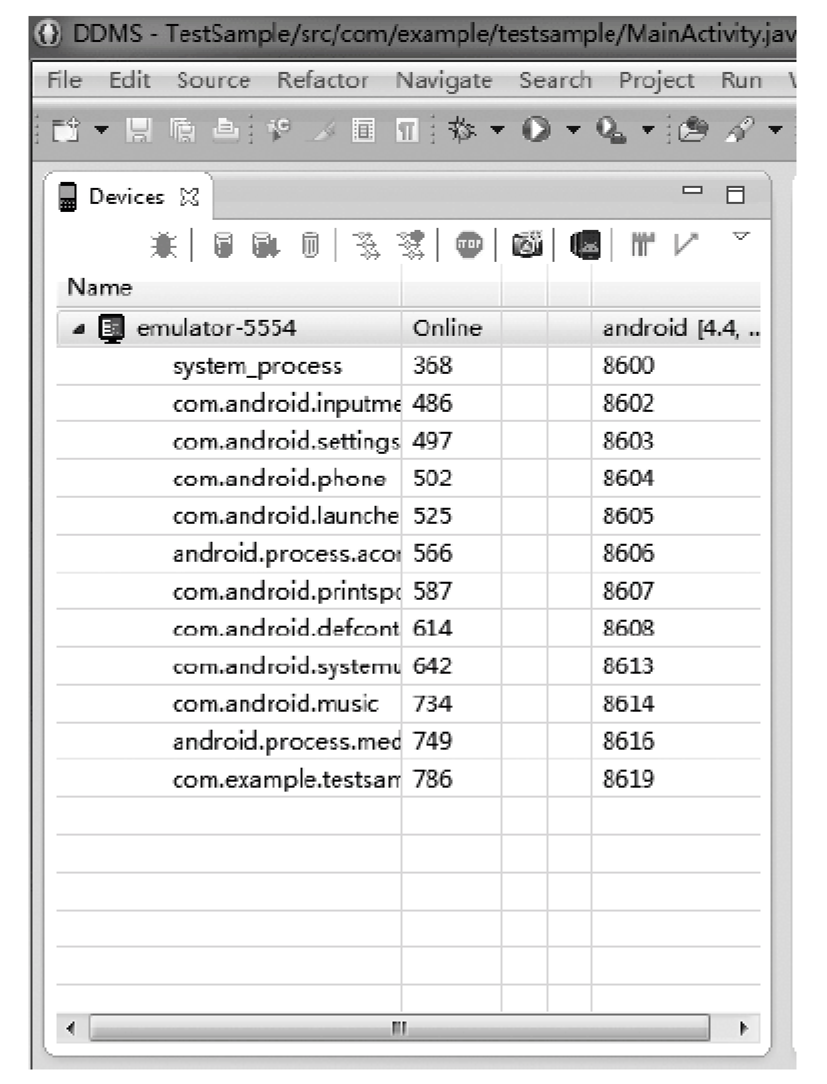


图 5.17 DDMS 监视器包含设备的连接端口分布

如图 5.17 所示,DDMS 监听第一个终端 App 进程的端口为 8600,App 进程将分配 8601,如果有更多终端或者更多 App 进程将按照这个顺序以此类推。DDMS 通过 8700 端口("base port")接收所有终端的指令。

(1) 在左上角,将能够找到处于运行状态的模拟器和连接的设备。

(2) 文件浏览器方便查看模拟器和设备上的文件(包括应用程序文件、目录和数据库),并且可以进行提取和添加。

(3) LogCat 窗口能够让用户监视 Android 日志控制台(LogCat)。这里显示 Log.i()、Log.e()和其他 Log 方法调用产生的消息。

(4) 可以查看每一个进程(堆和线程更新),也可以查看每一个线程,还可以终止进程。可以触发进程上的"垃圾回收(garbage collection)",并随后查看应用程序所使用的堆。

(5) 可以使用 Screen Capture(屏幕捕捉)按钮来捕捉模拟器和设备上的屏幕画面。

(6) 随时可以使用模拟器控制台,发送 GPS 消息、模拟来电或者 SMS 发送消息。

5.4.3 DDMS 具体功能

1. Device

如图 5.18 所示,在左上角可以看到标签为"Devices"的面板,这里可以查看到所有与 DDMS 连接的终端的详细信息,以及每个终端正在运行的 App 进程,每个进程最右边相对应的是与调试器连接的端口。因为 Android 是基于 Linux 内核开发的操作平台,同时也保留了 Linux 中特有的进程 ID,它介于进程名和端口号之间。

图 5.18 DDMS 监视器连接界面面板操作按钮

并且可以通过 Devices 面板进行一些操作,

例如 Debug the selected process、Update Threads、Update Heap、Stop Process 和 ScreenShot。并且调试选择的进程时，为其关联一个调试器，这样就可以在提供了源代码的情况下调试这个进程。

点拨　DDMS 调试过程中只能选择一个虚拟机实例或真实设备进行连接。

2. Emulator Control

通过这个面板的一些功能可以非常容易地使测试终端模拟真实手机所具备的一些交互功能，比如：接听电话，根据选项模拟各种不同网络情况，模拟接收 SMS 消息和发送虚拟地址坐标用于测试 GPS 功能等，如图 5.19 所示。

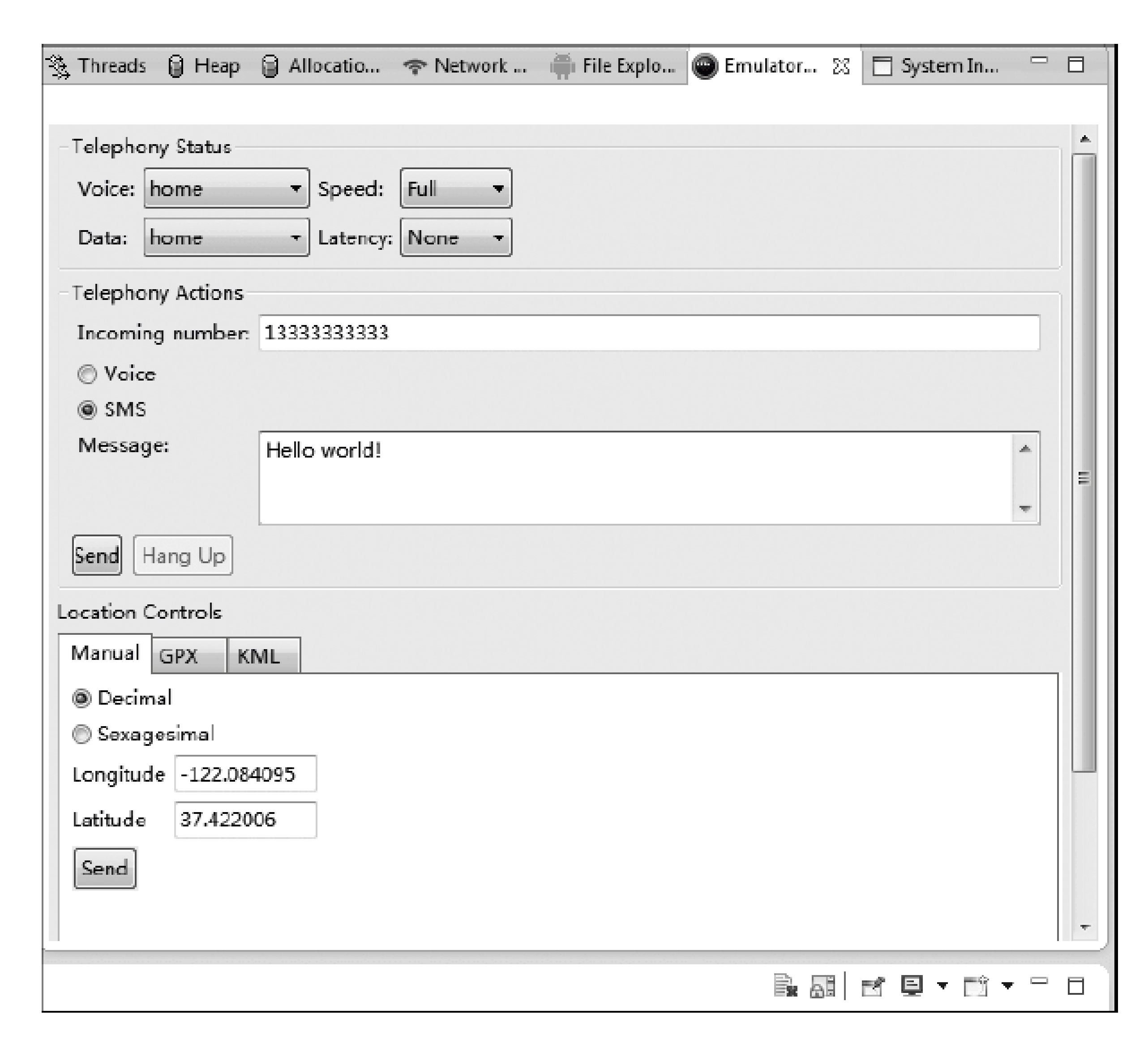


图 5.19　DDMS 监视器虚拟机控制界面

Telephony Status：通过选项模拟语音质量以及信号连接模式。

Telephony Actions：模拟电话接听和发送 SMS 到测试终端。

Location Controls：模拟地理坐标或者模拟动态的路线坐标变化并显示预设的地理标识，可以通过以下三种方式。

(1) Manual：手动为终端发送二维经纬坐标。

(2) GPX：通过 GPX 文件导入序列动态变化地理坐标，从而模拟行进中 GPS 变化的数值。

(3) KML：通过 KML 文件导入独特的地理标识，并以动态形式根据变化的地理坐标显示在测试终端。

5.4.4　进程监控

DDMS 非常有用的一个特性在于可以同进程进行交互。每一个 Android 应用程序都有一个独立的用户 ID 运行在模拟器中。

通过 DDMS 的 Devices 面板可以查看连接的模拟器实例，每一个都以其包名作为标识。

通过面板可以做到：

(1) 在 Eclipse 中关联并调试应用程序；

(2) 监视线程；

(3) 监视堆；

(4) 终止进程；

(5) 强制进行垃圾回收。

例如，通过图 5.20 可以看到包 com.android.inputmethod.latin 运行在模拟器上。

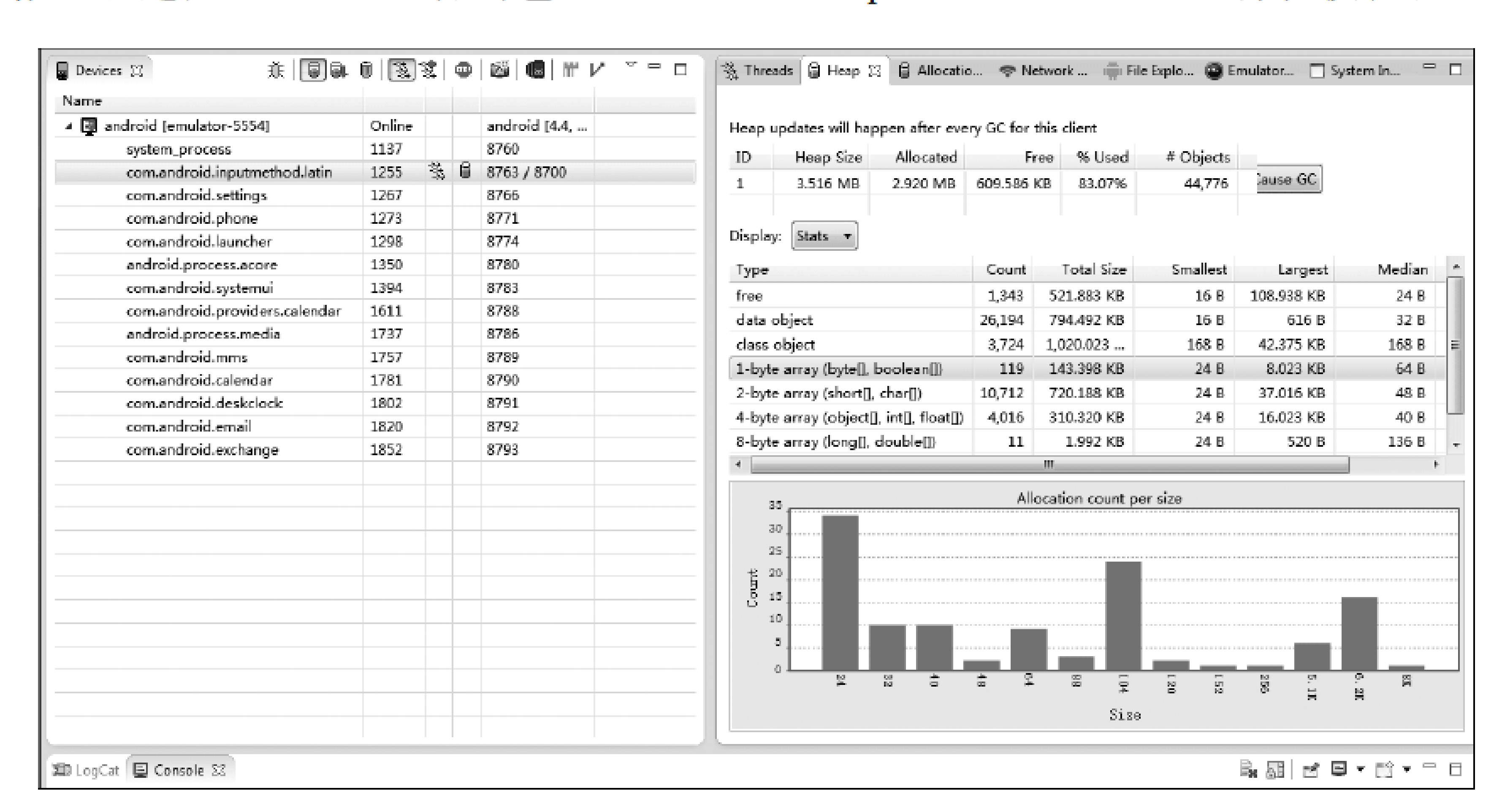


图 5.20 DDMS 调试过程中堆使用情况

选定感兴趣的进程，并同时选择 Update Threads 选项后，则在右边线程中可以查看运行时所包含的线程数目以及线程的详细信息，如图 5.21 所示。

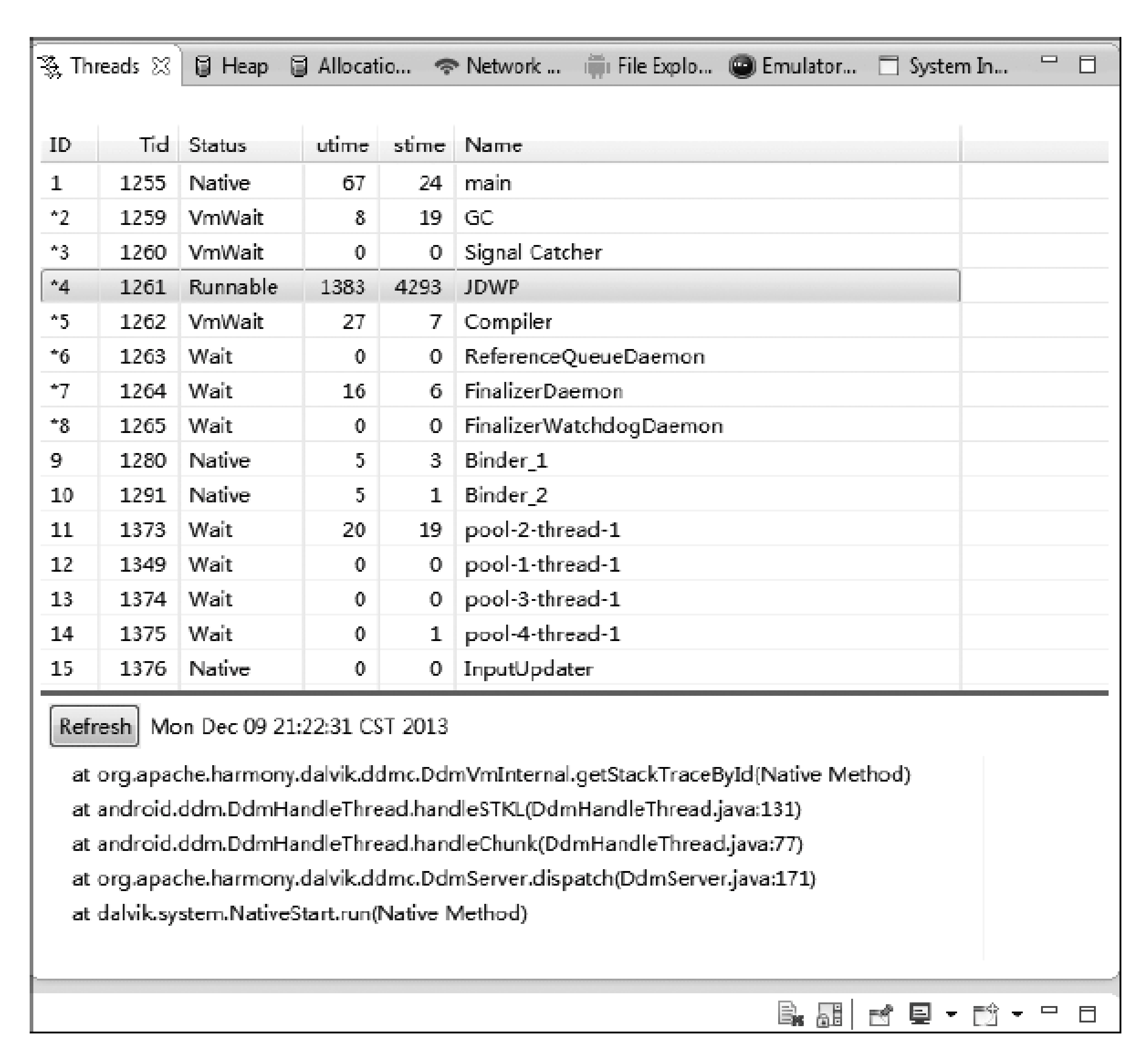
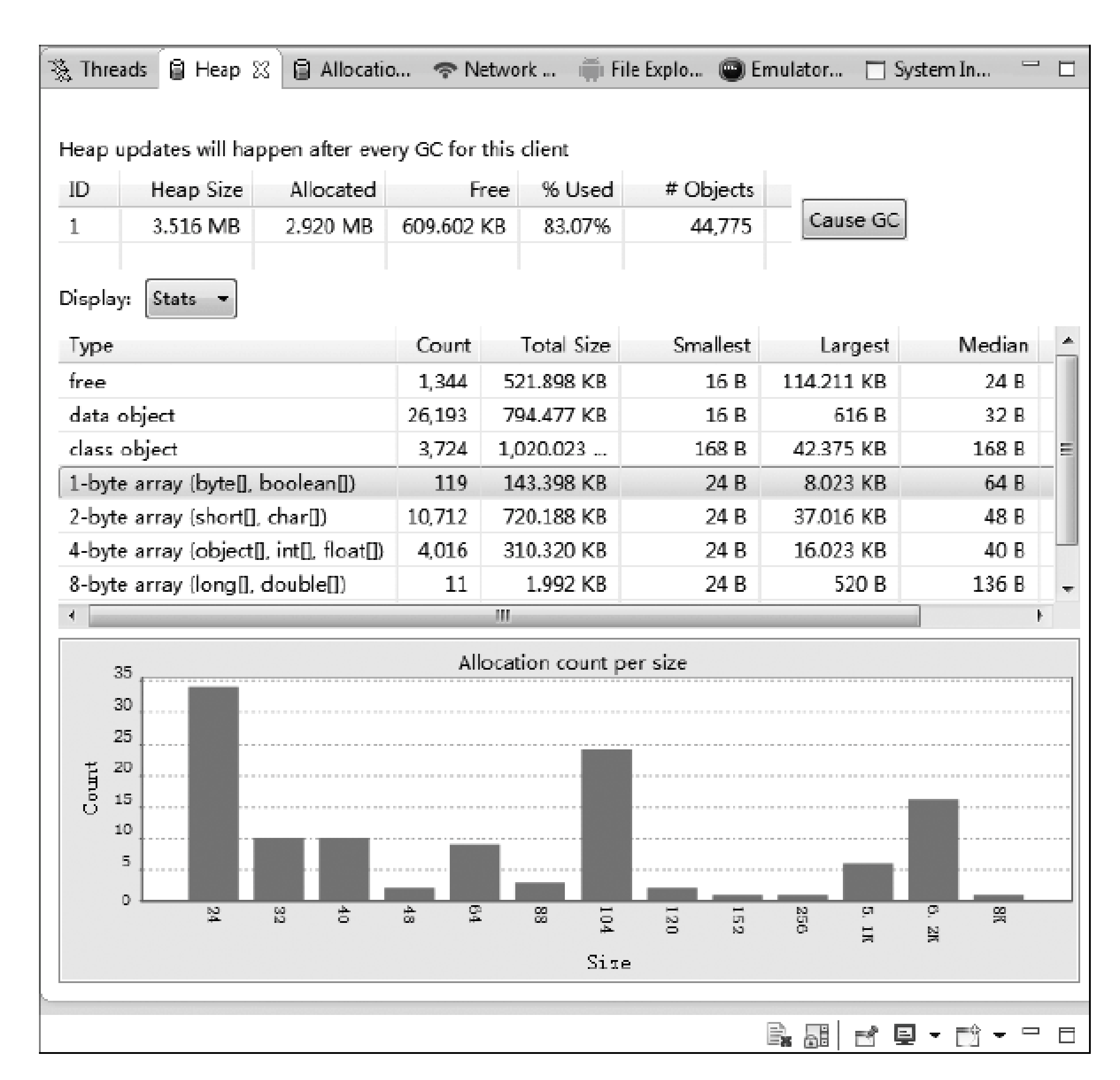


图 5.21 DDMS 调试过程中包的具体运行线程

当选定 Update Heap 选项后，在 Heap 中进行强制垃圾回收(GC)后，可查看包运行时所申请的堆大小和利用率，以及包运行时包含的不同变量数据的申请次数和占用的堆的大小情况。并通过柱状图来清晰地显示自己关心的数据在运行时占用堆大小的次数分布，如图 5.22 所示。



ID	Heap Size	Allocated	Free	% Used	# Objects
1	3.516 MB	2.920 MB	609.602 KB	83.07%	44,775

Type	Count	Total Size	Smallest	Largest	Median
free	1,344	521.898 KB	16 B	114.211 KB	24 B
data object	26,193	794.477 KB	16 B	616 B	32 B
class object	3,724	1,020.023 ...	168 B	42.375 KB	168 B
1-byte array (byte[], boolean[])	119	143.398 KB	24 B	8.023 KB	64 B
2-byte array (short[], char[])	10,712	720.188 KB	24 B	37.016 KB	48 B
4-byte array (object[], int[], float[])	4,016	310.320 KB	24 B	16.023 KB	40 B
8-byte array (long[], double[])	11	1.992 KB	24 B	520 B	136 B

图 5.22　DDMS 调试具体变量的堆申请分配详细信息

在 Allocation 面板中通过对运行进程进行空间申请跟踪，了解当前进程包含变量的内存申请的次数和所属线程 ID 等一些详细信息，如图 5.23 所示。

Threads　Heap　Allocatio...　Network ...　File Explo...　Emulator...　System In...

Stop Tracking　Get Allocations　Filter:　Inc. trace

Alloc Order		Allocated Class	Thread Id	Allocated in	Allocated in
5	290	byte[]	4	org.apache.harmon...	getThreadStats
6	56	java.nio.ByteArray...	4	java.nio.ByteBuffer	wrap
14	24	byte[]	4	dalvik.system.Nativ...	run
12	24	org.apache.harmo...	4	org.apache.harmon...	dispatch
10	24	org.apache.harmo...	4	android.ddm.Ddm...	handleREAQ
9	24	byte[]	4	dalvik.system.Nativ...	run
7	24	org.apache.harmo...	4	org.apache.harmon...	dispatch
4	24	org.apache.harmo...	4	android.ddm.Ddm...	handleTHST
3	24	byte[]	4	dalvik.system.Nativ...	run
1	24	org.apache.harmo...	4	org.apache.harmon...	dispatch
11	17	byte[]	4	android.ddm.Ddm...	handleREAQ
13	12	java.lang.Integer	4	java.lang.Integer	valueOf
8	12	java.lang.Integer	4	java.lang.Integer	valueOf
2	12	java.lang.Integer	4	java.lang.Integer	valueOf

图 5.23　DDMS 进程申请空间信息跟踪

点拨 DDMS 调试过程中能选择一个包或多个包进行调试，并且在调试中查看堆的使用状态时必须对调试的包进行垃圾强制回收后才能得到详细信息。建议单个包进行调试，这样调试过程清晰简洁，可具有针对性地改进程序。

5.4.5 使用文件浏览器

可以使用 DDMS 来查看并操作模拟器和设备上的 Android 文件系统。表 5.1 给出了 Android 文件系统中的一些重要文件。

表 5.1 Android 系统文件目录列表

目　录	说　明
/data/data/<package name>/	应用程序顶层目录
/data/data/<package name>/shared_prefs/	应用程序共享首选项目录
/data/data/<package name>/files/	应用程序文件目录
/data/data/<package name>/cache/	应用程序缓存目录
/data/data/<package name>/database/	应用程序数据库目录
/sdcard/dowmload/	用于存储浏览器下载图像
/data/app/	用于存储第三方 Android 程序文件

通过 DDMS 浏览 Android 文件系统如图 5.24 所示。

Threads　Heap　Allocatio...　Network ...　File Explo...　Emulator...　System In...

Name	Size	Date	Time	Permissions	Info
acct		2013-12-09	02:54	drwxr-xr-x	
cache		2013-12-09	02:54	drwxrwx---	
config		2013-12-09	02:54	dr-x------	
d		2013-12-09	02:54	lrwxrwxrwx	-> /sys/kernel/debug
data		2013-12-08	07:40	drwxrwx--x	
default.prop	116	1969-12-31	19:00	-rw-r--r--	
dev		2013-12-09	02:54	drwxr-xr-x	
etc		2013-12-09	02:54	lrwxrwxrwx	-> /system/etc
file_contexts	8870	1969-12-31	19:00	-rw-r--r--	
fstab.goldfish	953	1969-12-31	19:00	-rw-r-----	
init	175260	1969-12-31	19:00	-rwxr-x---	
init.environ.rc	919	1969-12-31	19:00	-rwxr-x---	
init.goldfish.rc	2979	1969-12-31	19:00	-rwxr-x---	
init.rc	19837	1969-12-31	19:00	-rwxr-x---	
init.trace.rc	1795	1969-12-31	19:00	-rwxr-x---	
init.usb.rc	3915	1969-12-31	19:00	-rwxr-x---	
mnt		2013-12-09	02:54	drwxrwxr-x	
proc		1969-12-31	19:00	dr-xr-xr-x	
property_contexts	2161	1969-12-31	19:00	-rw-r--r--	
root		2013-07-09	20:46	drwx------	
sbin		1969-12-31	19:00	drwxr-x---	
sdcard		2013-12-09	02:54	lrwxrwxrwx	-> /storage/sdcard
seapp_contexts	656	1969-12-31	19:00	-rw-r--r--	
sepolicy	74708	1969-12-31	19:00	-rw-r--r--	
storage		2013-12-09	02:54	drwxr-x--x	
sys		2013-12-09	02:54	dr-xr-xr-x	

图 5.24 DDMS 文件浏览器查看虚拟机文件列表

通过图 5.24 你还可以了解到系统文件的一些基本信息，包括文件名称、目录、文件大小、创建时间、操作权限以及简单描述信息。通过文件浏览器可以从模拟器或设备上复制文

件，向模拟器或设备复制文件以及删除模拟器或设备上的文件。

点拨　通过文件浏览器可以做到对文件的删除操作，对文件删除时要谨慎，删除后的文件没有办法进行恢复，容易造成严重的错误，如误删除系统重要文件。

5.4.6　模拟器控制

可以通过 DDMS 的 Emulator Control 选项卡来操作模拟器的实例，通过模拟器控制可以做到以下事情。

(1) 修改通话状态；

(2) 模拟语音通话；

(3) 模拟 SMS 接收；

(4) 发送位置坐标。

模拟器控制面板如图 5.25 所示。

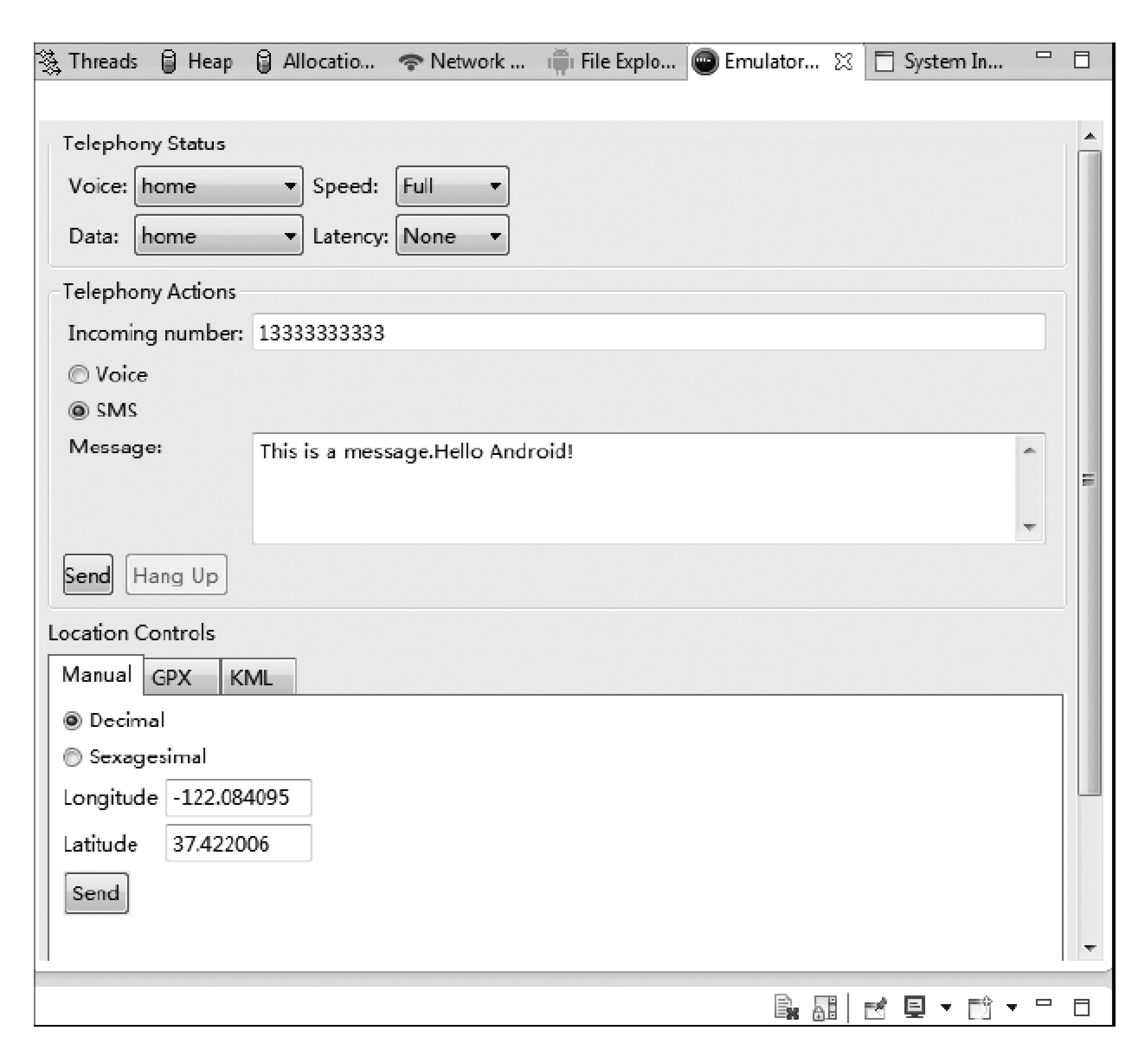


图 5.25　DDMS 虚拟机控制器界面

DDMS 提供稳定的向模拟器发送 SMS 的方法。要使用模拟器控制标签模拟发送 SMS，操作过程如下。

(1) 在 Telephony Actions 中选定 SMS；

(2) 输入模拟发送的电话号码，电话号码可以包含任意数字、“＋”和“＃”；

(3) 输入 SMS 消息；

(4) 单击 Send 按钮进行发送。

完成发送后在模拟器信息中会收到模拟发送的 SMS 信息，接收 SMS 可能存在中文识别问题。这是需要注意的地方。可以通过图 5.26 看到模拟器中的短信接收界面。

点拨　模拟机控制器可以模拟 SMS 发送，电话号码没有具体的限制，可以通过错误号码来检测开发程序的正确性。还可以通过 SMS 中发送中文来检测程序的兼容性。

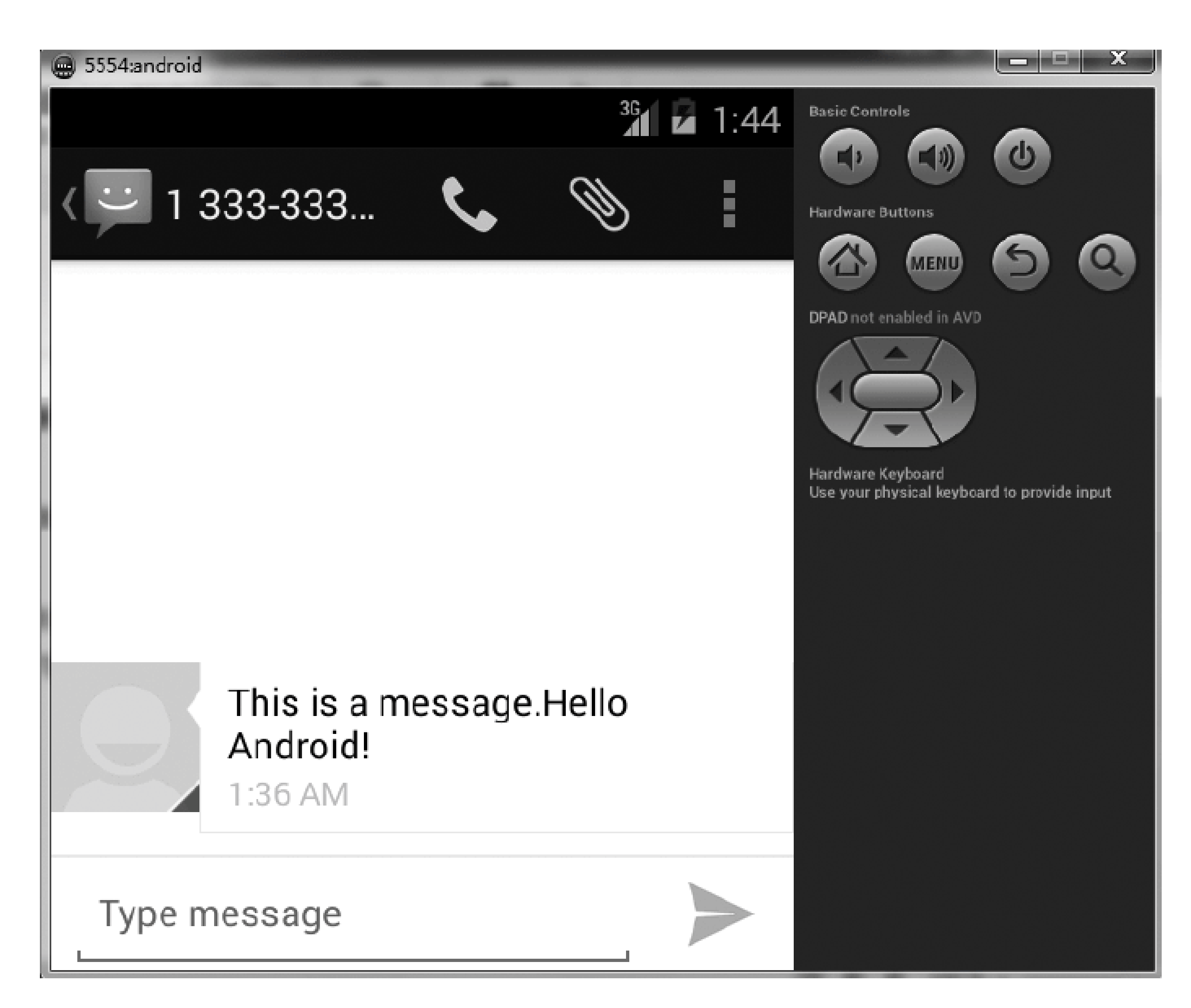


图 5.26 虚拟机接收 DDMS 模拟发送的信息

5.4.7 应用程序日志

DDMS 中结合了 LogCat 日志工具。可以通过 LogCat 日志记录来查看调试过程中的所有日志信息。这些信息包括 Debug、Information、Warning、Error 等信息。LogCat 默认是全部显示日志信息,通过自己选择可以查看自己关心的日志信息,如图 5.27 所示。

LogCat Console

Saved Filters

All messages (no filters)

Search for messages. Accepts Java regexes. Prefix with pid:, app:, tag: or text: to limit scope. verbose

L...	Time	PID	TID	Application	Tag	Text
W	?:??	?	-1		?	--------- beginning of /dev/log/main
E	12-10 01:25:21.570	38	38		logwrapper	executing /system/bin/e2fsck failed: No such file or director
E	12-10 01:25:21.600	41	41		logwrapper	executing /system/bin/e2fsck failed: No such file or director
I	12-10 01:25:23.290	45	45		qemu-props	connected to 'boot-properties' qemud service.
I	12-10 01:25:23.290	45	45		qemu-props	receiving..
I	12-10 01:25:23.290	45	45		qemu-props	received: dalvik.vm.heapsize=32m
I	12-10 01:25:23.320	45	45		qemu-props	receiving..
I	12-10 01:25:23.320	45	45		qemu-props	received: qemu.sf.lcd_density=240

图 5.27 LogCat 日志信息显示

还可以创建自定义过滤标签来显示仅与调试标记相关的日志信息。可以通过"+"按钮来添加过滤标签,这对应用程序调试过程相当有帮助,只显示相关的日志活动。

在调试过程中还可以通过 DDMS 进行截取模拟器或设备的屏幕显示。设备截屏对于调试来讲非常有用。

截取屏幕显示可以通过 DDMS 中的 Screen Capture 来进行屏幕截取,单击后会出现截取界面,当显示后查看是否为想截取的界面,确认后通过 Save 来保存截取的屏幕画面。DDMS 屏幕画面截图功能如图 5.28 所示。



图 5.28　DDMS 截图操作效果

小结

本章相对来说具有很强的操作性,从 Dalvik 虚拟机的工具介绍,我们看到了 Dalvik 虚拟机在程序开发过程中的调试分析作用,通过工具的使用对虚拟机内部的实现机制更加清晰,了解了如何对 Android 程序源码进行调试分析内存泄漏,如何对 Android 程序运行过程中生成的 trace 文件进行分析,如何使用 DDMS 工具对程序源码进行准确的跟踪调试分析。对于大多数从事 Android 程序开发的程序员来说,熟练运用这些工具,分析 Android 程序源码将变得得心应手,内存泄漏作为常常被程序员在编码过程中忽略的问题,资源使用后未释放,有时候只有当程序大量消耗内存卡死的时候才体现出来,通过对堆栈和程序运行过程中 trace 文件进行分析,可以准确定位程序中源码语句,有效地提高程序运行效率。通过 DDMS 工具可以详细地调试开发程序在内存使用方面的信息,可以通过这些信息进一步优化程序,有助于开发高效的应用程序。

第 6 章 Dalvik虚拟机执行流程详解

本章主要内容

✍什么是 Dalvik 虚拟机?

✍Dalvik 虚拟机包含哪些功能模块?

✍Dalvik 虚拟机在不同平台运行时从何开始?

✍Zygote 机制是如何实现的?

✍Zygote 机制在 Dalvik 虚拟机中充当怎样的角色?

✍Dalvik 虚拟机启动流程是怎样的?

Dalvik 虚拟机作为 Android 运行环境的核心组成,在其执行过程中需要各个模块的协调配合。本章简要介绍了 Dalvik 虚拟机的作用及组成,从 Dalvik 虚拟机运行流程着手,分析 Dalvik 虚拟机运行在不同平台的启动过程,着重分析了 Dalvik 虚拟机中必不可少的线程管理机制——Zygote 机制,剖析了 Zygote 在 Dalvik 虚拟机运行时所发挥的作用及其主要实现。对 Dalvik 虚拟机中运行的 apk 文件的生成过程进行了说明。最后,以运行应用程序为例,验证 Dalvik 虚拟机中各个模块的主要作用及相互关系,从全局的角度展示了 Dalvik 虚拟机的执行流程。

6.1 本章概述

Dalvik 虚拟机是 Android 中 Java 程序的运行基础。它是针对低内存的设备对 Java 虚拟机的优化,允许在同一台设备上一次运行多个虚拟机实例。每一个 Android 应用在底层都会对应一个独立的 Dalvik 虚拟机实例,其代码在虚拟机的解释下得以执行。Dalvik 虚拟机较接近底层的 Linux 操作系统,并且依赖其线程、内存等管理机制,可以高效地利用内存,并能在移动设备等这样低速的 CPU 上表现出较高的性能。

Java 应用程序转换为 Dex 文件后即可在 Dalvik 虚拟机中执行。Dex 文件是 Dalvik 虚拟机中专有的可执行文件格式,它整合优化了 class 文件,更适合于移动设备。Dalvik 虚拟机按照各部分的功能可以分为:线程管理、类加载、解释器、内存管理、即时编译、本地方法调用、反射机制、调试支撑几个部分。

线程管理:进程隔离和线程管理,每一个 Android 应用在底层都会对应一个独立的 Dalvik 虚拟机实例,所有的 Android 应用的线程都对应一个 Linux 线程,进程管理依赖于 Zygote 机制。

类加载:解析 Dex 文件并加载 Dalvik 字节码。

解释器:根据自身的指令集 Dalvik ByteCode 解释字节码。

内存管理：分配系统启动初始化和应用程序运行时需要的内存资源。

即时编译(Just-In-Time,JIT)：在解释时动态地编译程序，以缓解解释器的低效工作。

本地方法调用(Java Native Interface,JNI)：一套编程框架标准接口，允许 Java 代码和本地代码互相调用。

反射机制实现模块：允许程序在运行时透过 Reflection API 取得任何一个已知名称的类的内部信息，包括其描述符、超类、实现的接口，也包括属性和方法等所有信息，并可于运行时改变属性内容或调用内部方法。

调试支撑模块：Dalvik VM 支持许多常见开发环境下的代码级调试，任何允许 JDWP 下远程调试的工具都可以使用，其支持的调试器包括 jdb、Eclipse、IntelliJ 和 JSwat。

应用程序封装到 Dex 文件后，通过 Zygote 机制为其创建新的虚拟机，调用类加载将程序加载到 Dalvik 虚拟机中，在内存管理等模块的配合下由解释器进行取指执行。各个模块具体分析详见相应章节。

6.2　Dalvik 虚拟机的入口点介绍

在 Dalvik 虚拟机中，每一个应用程序均对应一个虚拟机实例，因而在运行应用程序之前需要创建虚拟机，由虚拟机负责应用程序的运行。在 Android 中，Dalvik 虚拟机既可以在 x86 平台上作为应用程序运行，也可以在手机等移动设备的 ARM 平台上运行，在不同平台运行时，虚拟机的入口点略有不同，以下主要分析 Dalvik 虚拟机这两种运行方式的入口点。

6.2.1　Dalvik 虚拟机在 x86 平台运行的入口点

将 Dalvik 虚拟机作为应用程序运行在 x86 平台时，通过调用 main 函数启动。其 main 函数在 Dalvik 虚拟机源码目录 dalvik/dalvikvm/Main.c 文件中，是 Dalvik 虚拟机作为 Linux 下 x86 应用程序运行时的入口。在其中调用 JNI_CreateJavaVM()函数来创建虚拟机。主要代码如下。

代码清单 6.1　dalvik/dalvikvm/Main.c：main()源代码

```
int main(int argc, char* const argv[]){
    JavaVM* vm=NULL;
    JNIEnv* env=NULL;
    JavaVMInitArgs initArgs;
        /**启动虚拟机。当前线程成为主线程的虚拟机。*/
    if (JNI_CreateJavaVM(&vm, &env, &initArgs) < 0) {
        fprintf(stderr, "Dalvik VM init failed (check log file)\n");
        goto bail;
}
```

Dalvik 虚拟机作为 Linux 下 x86 应用程序运行，主要是用于开发和调试使用。我们知道，在一个资源有限的平台里，如手机等移动设备，进行开发和调试，都是一件不容易的事情，需要花费很多时间。而利用目前 x86 的平台，可以使用 x86 大量的工具和资源，可以很大程度上提高开发效率，同时也使调试功能更加容易。在 Dalvik 虚拟机进行开发新功能

时，可以先在 x86 的平台上运行和调试通过，然后再编译在 ARM 平台运行。当 Dalvik 虚拟机运行在手机平台时，它的入口点并不是从该 main 函数开始。

6.2.2 Dalvik 虚拟机运行在 ARM 平台的入口点

Dalvik 虚拟机运行在 ARM 平台时，是从初始化进程加载的服务 Zygote 开始的。Zygote 进程介绍见 6.3 节。

Zygote 进程通过调用 startVm 来创建虚拟机。在这个函数中主要通过调用 JNI_CreateJavaVM()来创建虚拟机。主要代码如下。

代码清单 6.2 android/frameworks/base/core/jni/AndroidRuntime.cpp：startVm()源代码

```
int AndroidRuntime::startVm(JavaVM** pJavaVM, JNIEnv** pEnv)
{
        ......
    if (JNI_CreateJavaVM(pJavaVM, pEnv, &initArgs) < 0) {
        LOGE("JNI_CreateJavaVM failed\n");
        goto bail;
    }
        ......
}
```

Dalvik 虚拟机的两个入口点最后均通过调用 JNI_CreateJavaVM()函数来完成虚拟机的创建。JNI_CreateJavaVM 所完成的主要工作如下。

(1) 检查 JNI 版本是否正确；

(2) 解析命令行参数，初始化 JNIEnv 和 JavaVM 全局变量；

(3) 初始化全局变量 gDvm；

(4) 调用 dvmStartup 初始化虚拟机的各个模块，包括初始化垃圾收集器、类加载器、字节码校验模块和解释器等，完成各模块的初始化后创建一个线程；

(5) 装载 Dalvik 虚拟机运行时核心类库并校验字节码。

JNI_CreateJavaVM 函数的实现在文件 dalvik/vm/Jni.c 里。主要代码如下。

代码清单 6.3 dalvik/vm/Jni.c：JNI_CreateJavaVM()源代码

```
jint JNI_CreateJavaVM(JavaVM* * p_vm, JNIEnv * * p_env, void* vm_args) {
    const JavaVMInitArgs * args= (JavaVMInitArgs * ) vm_args;
    ...
    /**初始化 VM. * /
    gDvm.initializing= true;
    std::string status=
    dvmStartup(argc, argv.get(), args->ignoreUnrecognized, (JNIEnv * )pEnv);
    gDvm.initializing= false;
    if (!status.empty()) {
        free(pEnv);
        free(pVM);
```

```
        LOGW("CreateJavaVM failed: %s", status.c_str());
        return JNI_ERR;
    }
    /**成功！ Return stuff to caller.*/
    dvmChangeStatus(NULL, THREAD_NATIVE);
    *p_env=(JNIEnv*)pEnv;
    *p_vm=(JavaVM*)pVM;
    LOGV("CreateJavaVM succeeded");
    return JNI_OK;
}
```

如代码所示，JNI_CreateJavaVM()函数通过调用 dvmStartup 函数完成虚拟机的初始化。

6.2.3　Dalvik 虚拟机的初始化

在不同平台运行的 Dalvik 虚拟机，其入口点虽不同，但它的初始化均调用 dvmStartup 函数来完成，如图 6.1 所示，在这个函数中，Dalvik 虚拟机中各个模块进行了初始化。

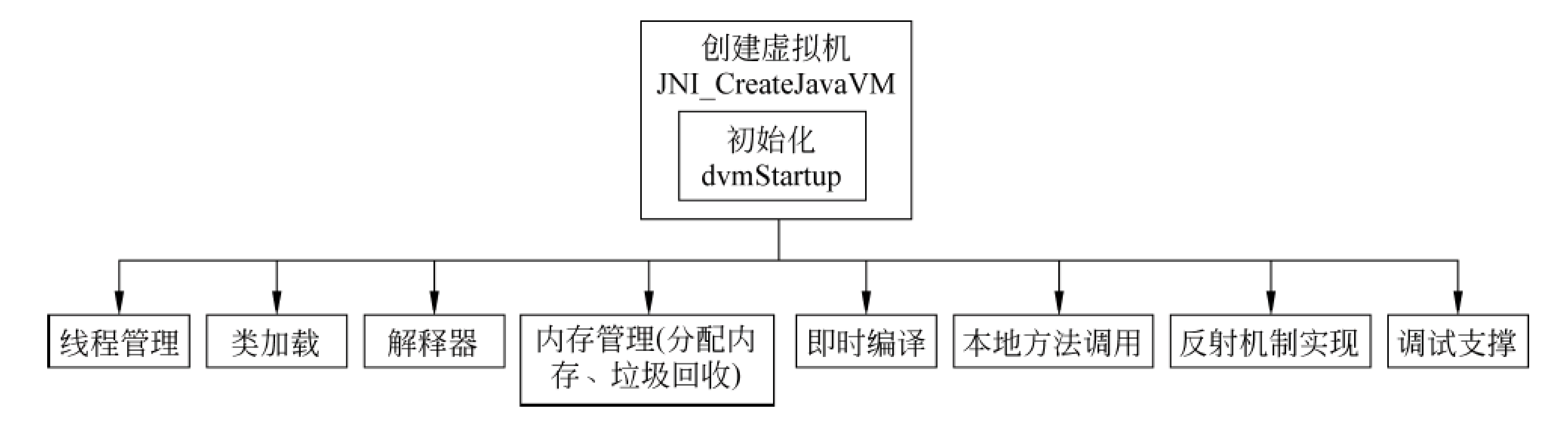


图 6.1　Dalvik 虚拟机的初始化模块图

6.3　Zygote 进程

Zygote 是 Android 系统应用中一个相当重要的进程，所有运行应用程序的虚拟机进程都是由 Zygote 创建的。Zygote 本身是一个 Native(参见 JNI 本地调用机制)的应用进程，与驱动、内核等均无关系。Zygote 进程是由 init 进程根据 system/core/rootdir/init.rc 文件中的配置项创建的，init 进程是系统启动后运行在用户空间的首个进程。init 进程启动完系统运行所需的各种 Daemon 线程(Java 将线程分为 User 线程和 Daemon 线程两种，其中 Daemon 线程即守护线程，运行在程序后台，用来为 User 线程提供某些服务)后，启动 Zygote 进程。Zygote 进程启动后，Android 的应用程序都由 Zygote 进程启动运行。Zygote 主要负责：

(1) 启动系统服务 SystemServer 进程；

(2) 创建子进程运行 Android 应用程序。

通过复制自身快速提供虚拟机实例来执行 Android 应用程序，Zygote 进程能为应用程序提供一个高效的运行环境，在应用程序运行时，有效地减少系统负担，提高设备的利用率，

加长设备的使用时间，使得应用程序在有限的资源下有更快的运行响应速度。

Zygote 是 Android 系统的主要特征，它通过 COW(copy_on_write)方式对运行在内存中的进程实现了最大程度的复用，并通过库共享有效地降低了内存的使用量。一般说来，复制内存的开销非常大，因此创建的 Zygote 子进程直接共享父进程的内存空间，而当需要修改共享内存中的信息时，子进程才会将父进程中的相关内存信息复制到自身的内存空间，并进行修改，这就是 COW 技术。

Dalvik 虚拟机中 Zygote 主要负责线程管理，与其他模块之间的关系如图 6.2 所示(注：图中较细的箭头代表流程，较粗的箭头代表依赖关系，下文图中的箭头与此规定相同)。

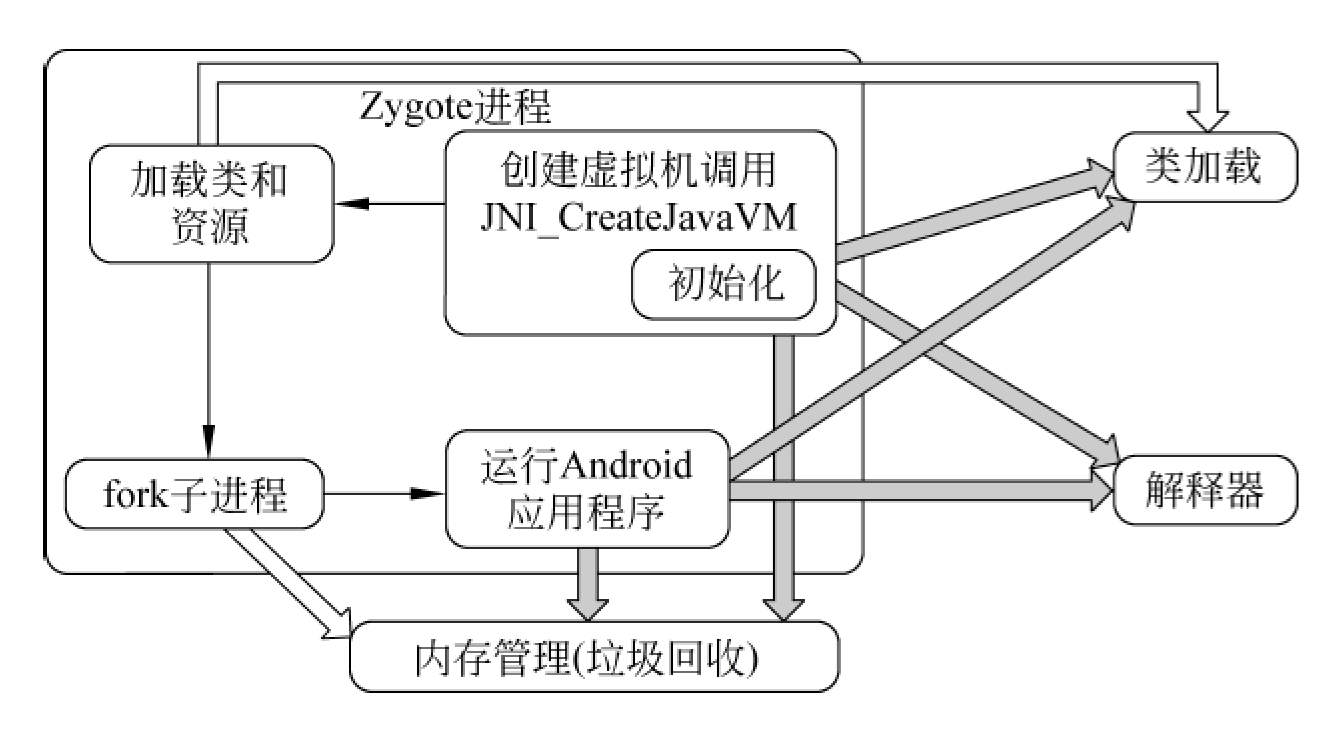


图 6.2　Zygote 与其他模块关系

在 Android 中，每个应用程序运行在各自的 Dalvik 虚拟机实例中，每一个虚拟机实例都是一个独立的进程空间。Android 应用程序运行在各自独立的进程空间由各自的用户控制，可以最大程度确保应用的安全和运行的独立性。

Zygote 是一个虚拟机进程，也是虚拟机实例的孵化器。它在系统启动时产生，完成虚拟机的初始化，库的加载，预制类库和初始化等操作。每当系统要求执行一个 Android 应用程序，Zygote 就会 fork 出一个子进程来执行该应用程序。Zygote 首先会孵化出 system_server 进程(Android 绝大多系统服务的守护进程，它会监听 socket 等待请求命令)，当系统需要一个新的虚拟机实例时，Zygote 会迅速复制自身，以最快的速度提供给系统。对于一些只读的系统库，所有虚拟机实例都和 Zygote 共享一块内存区域，可以有效地节省内存开销。Zygote 与其创建的子进程之间资源共享关系，如图 6.3 所示。

点拨　Zygote 进程运行时，会初始化 Dalvik 虚拟机，并启动它。Android 应用程序是由 Java 编写的，它们不能直接以本地进程的形态运行在 Linux 上，只能运行在 Dalvik 虚拟机中。并且，每个应用程序都运行在各自的虚拟机中，应用程序每次运行都要重新初始化并启动虚拟机，这个过程会耗费相当长时间，是拖慢应用程序的原因之一。因此，在 Android 中，应用程序运行前，Zygote 进程通过共享已运行的虚拟机的代码与内存信息，缩短应用程序运行所耗费的时间。Zygote 进程启动时先将应用程序要使用的 Android Framework 中的类与资源加载到内存中，并组织形成所用资源的链接信息。新运行的 Android 应用程序在使用所需资源时不必每次重新形成资源的链接信息，这会节省大量时间，提高程序运行速度。

Zygote 启动后，会初始化并运行 Dalvik 虚拟机，而后将需要的类与资源加载到内存中。

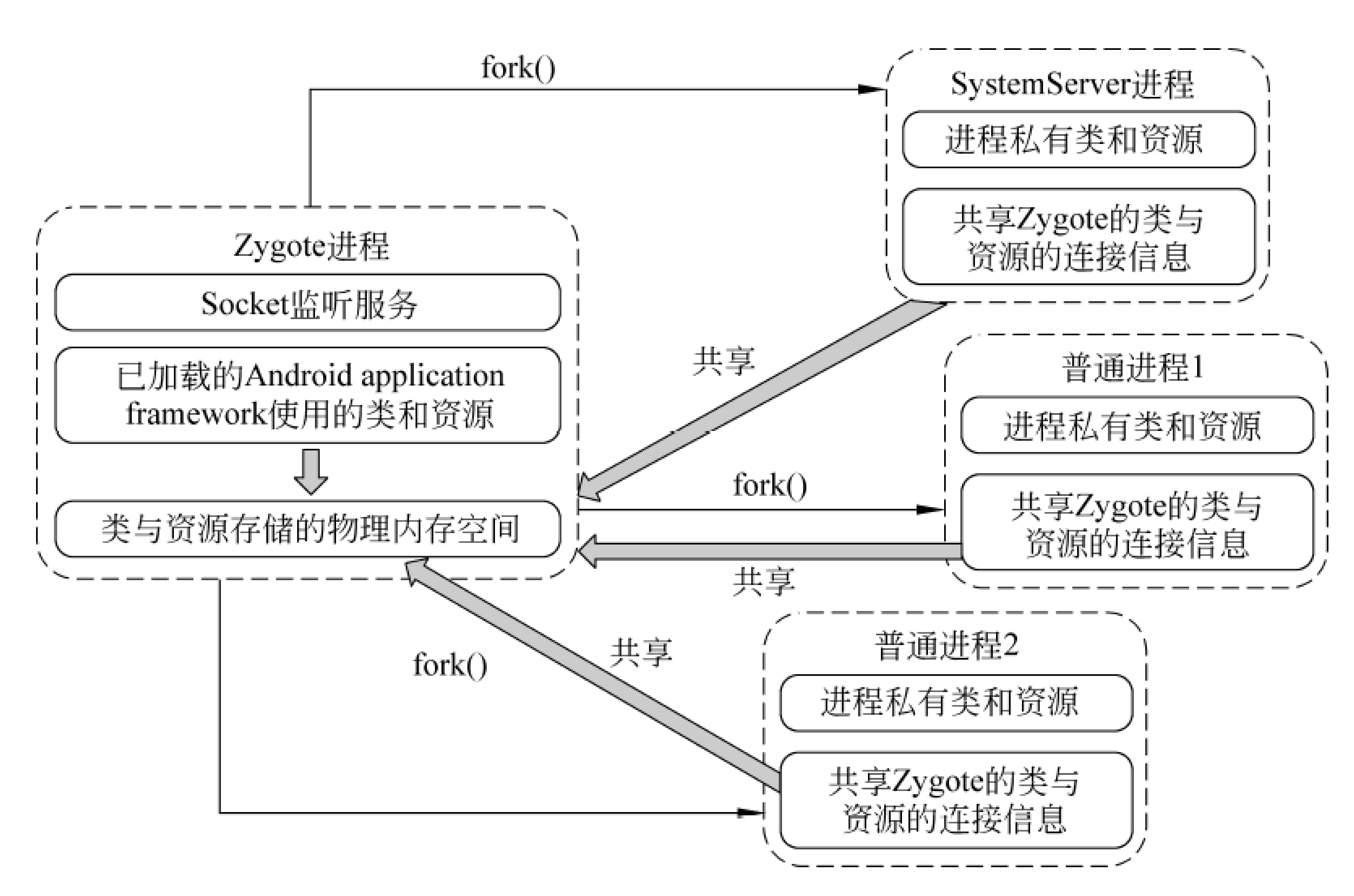


图 6.3　Zygote 进程创建子进程的内存共享示意图

当接收到应用程序 A 的启动请求时，Zygote 调用 fork()创建出 zygote'子进程，接着 Zygote'子进程动态加载并运行 Android 应用程序 A；运行应用程序 A 时，会使用 Zygote 已经初始化并启动运行的 Dalvik 虚拟机代码，通过使用已加载至内存中的类与资源来加快运行速度。

由此，Zygote 工作流程如下。

(1) 系统 init 进程创建 Zygote 进程，通过执行 app_process 程序，开启 Zygote 进程。

(2) app_process 生成 AppRuntime 对象，分析其主函数传递过来的参数，传递给 AppRuntime 对象，调用对象的 start 方法，在 start 中完成了以下三件事。

① 调用 startVm 注册虚拟机。在其中通过调用 JNI_CreateJavaVM()创建虚拟机。

② 调用 startReg 注册 JNI 函数。注册虚拟机要使用的 JNI 函数，这样运行在虚拟机中的 Java 类就可以调用本地函数了。

③ 调用 ZygoteInit 类的 main 函数，运行 ZygoteInit 类(位于 framework/base/core/java/com/androld/internal/os/ZygoteLnit.java)。

如图 6.4 所示 init 进程启动 Zygote 进程。

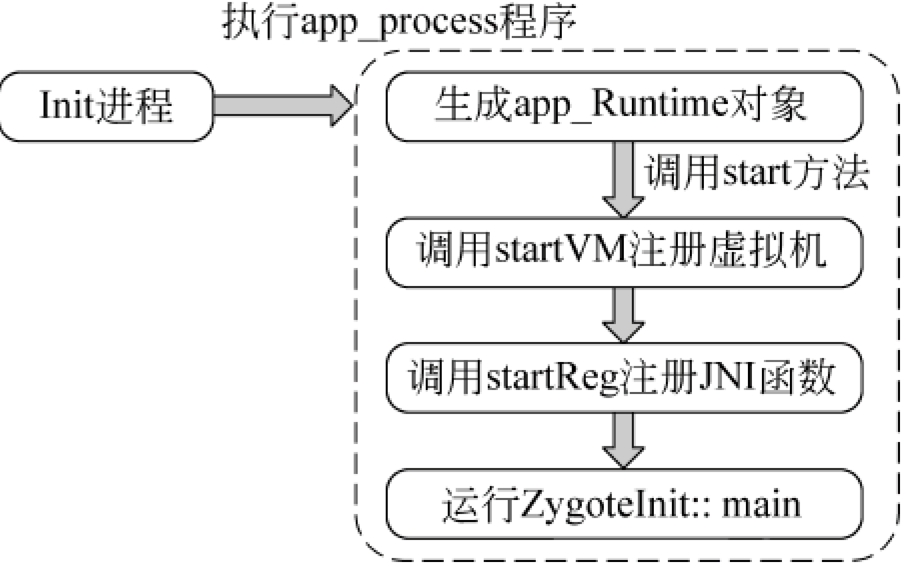


图 6.4　启动 Zygote 进程流程图

(3) ZygoteInit 是 Zygote 的 main 函数入口，是 Zygote 的核心类，完成了 Zygote 的职责，其执行流程如下。

① 调用 registerZygoteSocket 函数创建了一个 socket 接口，绑定 socket 套接字，接受新的 Android 应用程序运行请求。

② 调用 preloadClasses 和 preloadResource 函数加载 Android application framework 使用的类和资源。

③ 调用 startSystemServer 函数来启动 SystemServer 组件。在 startSystemServer 中

调用 forkSystemServer()来分裂出一个名为"system_server"的进程(这个过程在虚拟机中实现),这个进程最终会调用 com. android. server. SystemServer 的 main 函数,启动各项系统服务,并最终将调用线程加入到 Binder 通信系统。system_server 是系统 Service 所驻留的进程,该进程是 framework 的核心,一旦 system_server 进程退出,就会导致 Zygote 退出并重启。

④ Zygote 从 startSystemServer 函数返回后,最终调用 runSelectLoopMode 函数进入一个无限循环,监听 socket 接口等待新的应用程序请求。当新的应用程序请求到来时,在 runSelectLoopMode 中会接收连接的对象 ZygoteConnection(ZygoteConnection 是用来进行套接口连接管理及其参数解析,其他 Actvitiy 建立进程请求是通过套接口发送命令参数给 Zygote),通过调用 ZygoteConnection 的 runOnce()函数来运行应用程序。在 runOnce()中读取 system_server 发送过来的参数,调用 Dalvik 虚拟机中的 forkAndSpecialize()函数创建一个子进程用于运行应用程序,当应用程序运行完毕后,子进程退出。Zygote 继续监听等待新的应用程序请求。

图 6.5 表示 ZygoteInit 执行过程。

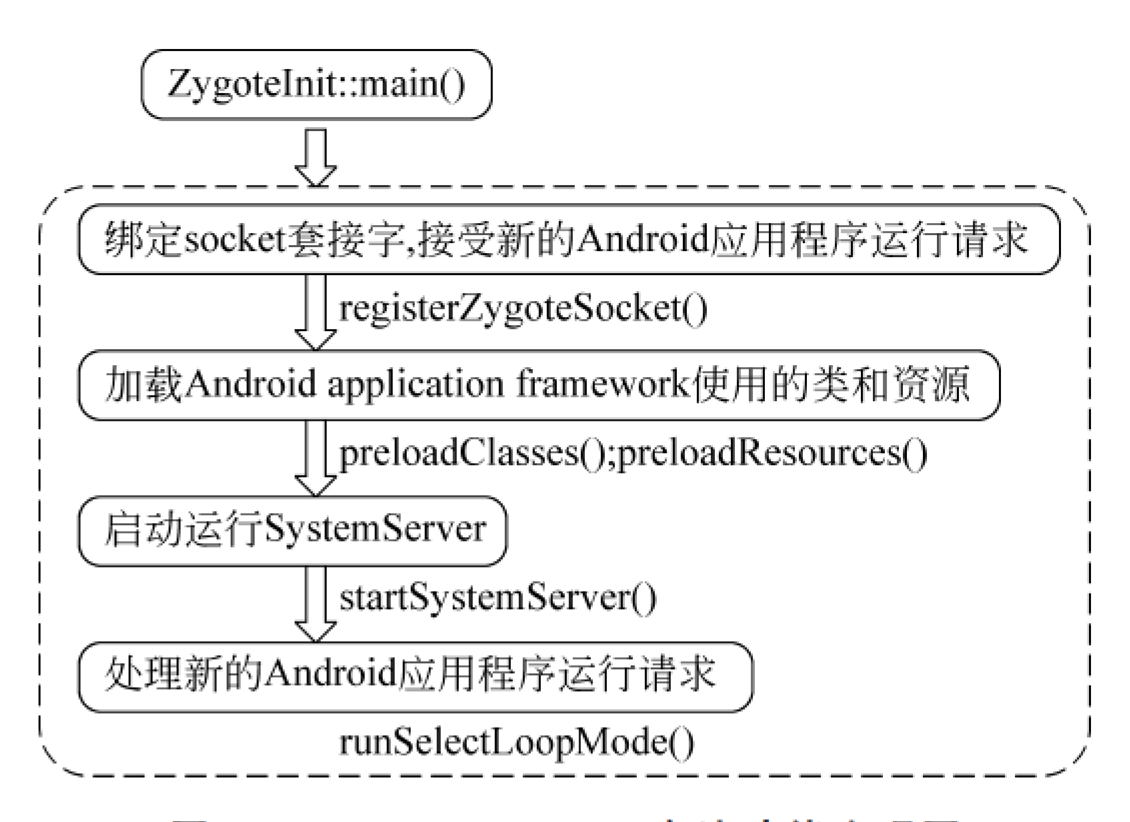


图 6.5　ZygoteInit:main 方法功能实现图

在 Dalvik 虚拟机中,Zygote 除了可以创建上述的 system_server 子进程外,还可以创建另两种进程,其创建三种进程的方式可用图 6.6 来描述。

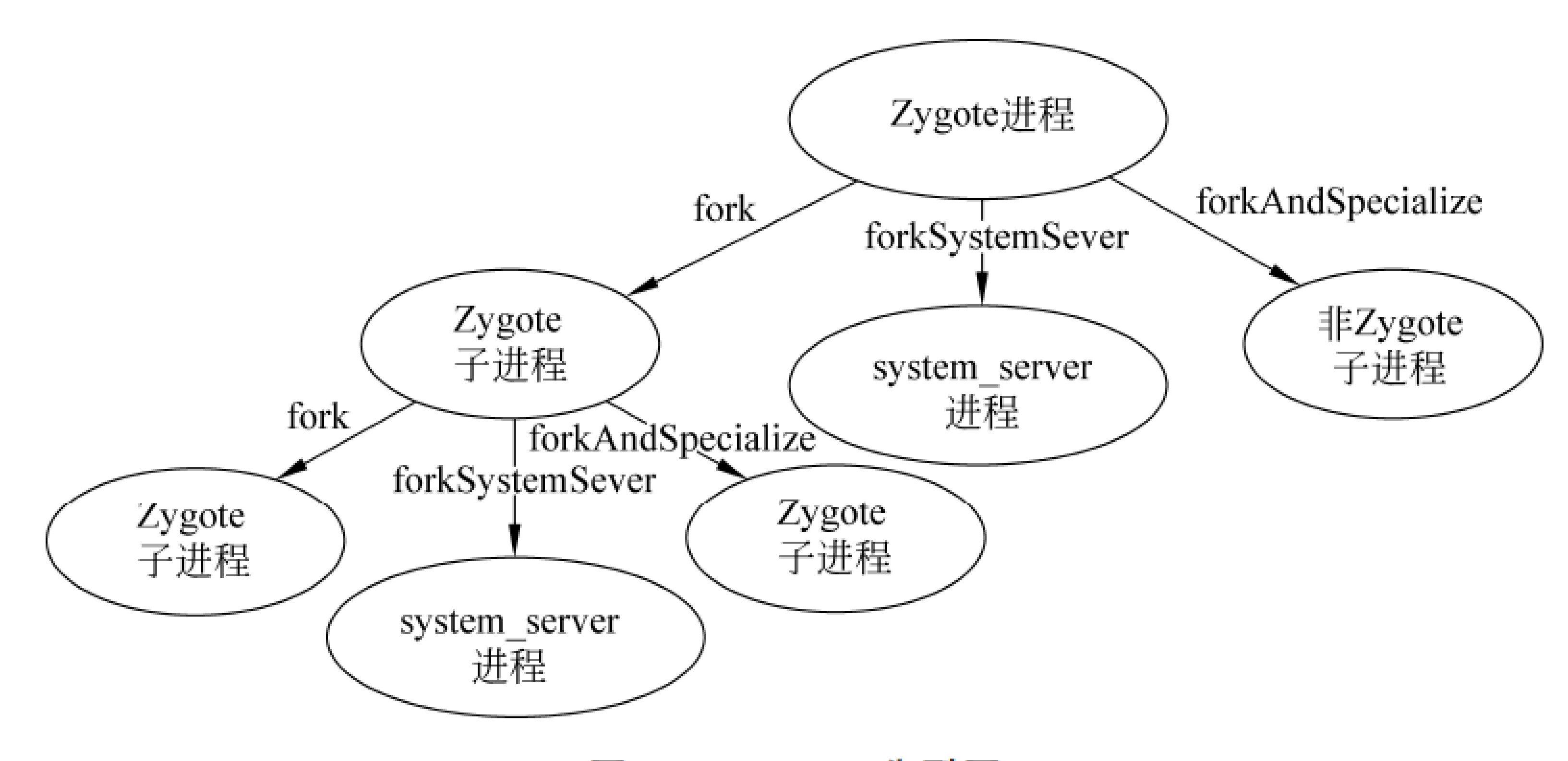


图 6.6　Zygote 分裂图

点拨　在 Dalvik 虚拟机中，Zygote 提供访问 Dalvik 虚拟机的 Zygote 接口，它包装了 Linux 系统的 fork 函数，用于建立一个新的虚拟机实例进程。Socket 套接字相关内容可查看 Java 进程通信相关知识。Android 中有许多相关的系统服务进程，这点在相关 Android 系统的书中均有介绍，故不再赘述。

在 Dalvik 虚拟机中，Zygote 机制的三种创建进程的方法根据 JNI 机制均对应到一个本地方法，位于 dalvik/vm/native/dalvik_system_Zygote. cpp 文件中，三个创建进程的静态方法分别如下。

(1) fork()，创建一个 Zygote 进程，其实现对应的函数为 Dalvik_dalvik_system_Zygote_fork()，代码如下。

代码清单 6.4　dalvik/vm/native/dalvik_system_Zygote. cpp：Dalvik_dalvik_system_Zygote_fork()源代码

```
static void Dalvik_dalvik_system_Zygote_fork(const u4* args, JValue* pResult)
{
    pid_t pid;
    /**判断当前虚拟机是否支持 zygote*/
    if (!gDvm.zygote) {
        dvmThrowIllegalStateException(
            "VM instance not started with -Xzygote");
        RETURN_VOID();
    }
    /**判断堆创建是否成功*/
    if (!dvmGcPreZygoteFork()) {
        LOGE("pre-fork heap failed");
        /**不成功,停止虚拟机*/
        dvmAbort();
    }/**设置信号机制*/
    setSignalHandler();
    /**记录日志信息*/
    dvmDumpLoaderStats("zygote");
    pid=fork();
#ifdef HAVE_ANDROID_OS
    if (pid==0) {
        /* child process */
        /**子进程 */
        extern int gMallocLeakZygoteChild;
        gMallocLeakZygoteChild=1;
    }
#endif
    RETURN_INT(pid);
}
```

fork()方法会重新产生一个新 Zygote 进程，新的子进程复制了父亲进程的资源，包括内存的内容、task_struct 内容，在复制过程中，子进程复制了父进程的 task_struct、系统堆

栈空间和页面表，而当子进程改变了父进程的变量的时候，会通过写时复制(copy_on_write)的手段为所涉及的页面建立一个新的副本，新的 Zygote 进程可以再产生子进程。

点拨 fork()函数通过系统调用创建一个与原来进程几乎完全相同的进程，也就是两个进程可以做完全相同的事，但如果初始参数或者传入的变量不同，两个进程也可以做不同的事。一个进程调用 fork()函数后，系统先给新的进程分配资源，例如存储数据和代码的空间。然后把原来的进程的所有值都复制到新的进程中，只有少数值与原来的进程的值不同。相当于克隆了一个自己。

Dalvik_dalvik_system_Zygote_fork()函数执行流程如图 6.7 所示。

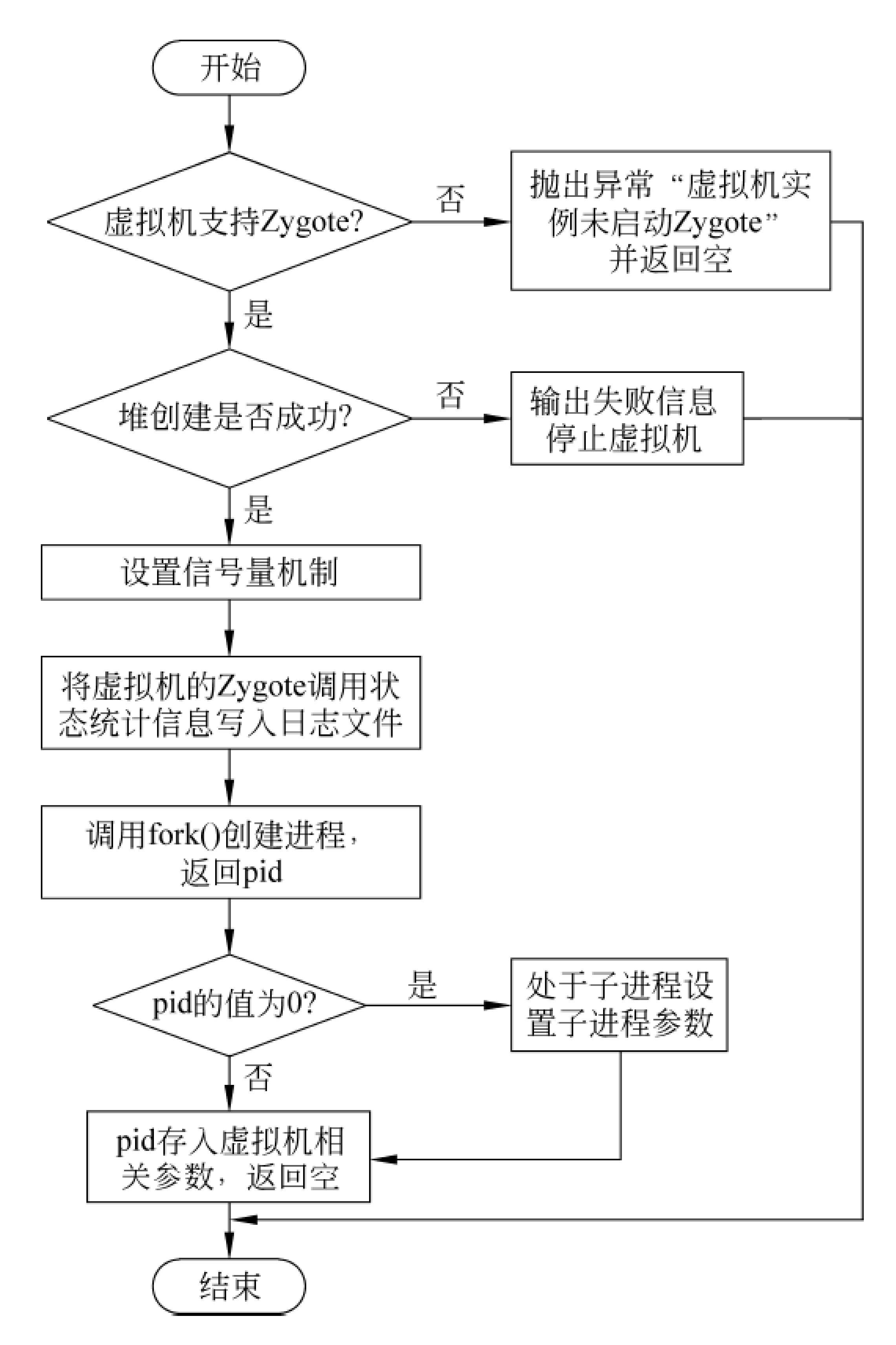


图 6.7 Dalvik_dalvik_system_Zygote_fork()函数流程图

在 Dalvik_dalvik_system_Zygote_fork()函数中直接调用了 fork()产生新的 Zygote 进程，新进程复制父进程的资源，新的 Zygote 进程的虚拟机参数中支持 Zygote 标记参数 gDvm.zygote 也为 true，因而可以再次产生出子进程。

在函数中调用了一次 fork()函数，得到的返回值 pid 有三个，这是因为：在 fork 函数执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork 函数返回 0，在父进程中，fork 返回新创建子进程的进程 ID；如果创建失败则返回负值。通过 fork 的返回值可以判断进程创建是否成功以及当前进程是子进程还是父进程。

（2）forkSystemServer()，创建一个系统服务进程，其实现对应的函数是 Dalvik_dalvik_system_Zygote_forkSystemServer()，代码如下。

代码清单 6.5　dalvik/vm/native/dalvik_system_Zygote. cpp：Dalvik_dalvik_system_Zygote_forkSystemServer()源代码

```
static void Dalvik_dalvik_system_Zygote_forkSystemServer(
        const u4* args, JValue* pResult)
{
    pid_t pid;
      /**根据参数，fork一个子进程*/
    pid= forkAndSpecializeCommon(args, true);

    /**Zygote进程检查子进程是否已经死掉 */
    if (pid> 0) {
        int status;
        LOGI("System server process %d has been created", pid);
        /**保存 system_server 的进程 id*/
        gDvm.systemServerPid= pid;
      /**在这里，如果 system server 进程已经崩溃，但是它并不能引起注意，因为还没
           * 有公布它的 pid，因此，这里检查 system server 是否退出，是为了确保准确。
           */
        if (waitpid(pid, &status, WNOHANG)==pid) {
              /**如果 system_server 退出了，Zygote 直接 kill 掉自己*/
            LOGE("System server process %d has died. Restarting Zygote!", pid);
            kill(getpid(), SIGKILL);
        }
    }
    RETURN_INT(pid);
}
```

forkSystemServer()的子进程不是 Zygote 进程，调用 forkSystemServer()创建的 system_server 子进程会一直驻留在系统中，一旦此进程退出，父进程 Zygote 也会被终止。系统 init 进程通过重启 Zygote 进程，使得 Zygote 进程重新启动 system_server 进程。

在 ZygoteInit 中使用 startSystemServer()来启动系统服务，startSystemServer()方法通过调用 Dalvik 虚拟机的 dalvik_system_Zygote. cpp 文件中的 forkSystemServer()方法来 fork “system_server”进程。与创建运行普通 Android 应用程序的进程不同，system_server 是必须运行的，因此在 forkSystemServer()方法中必须检查生成的 system_server 进程是否退出。

函数流程如图 6.8 所示。

当 Zygote 创建 system_server 后，在函数结束之前需要检查 system_server 进程是否退出，若发现其退出，Zygote 就会 kill 自身，由此可见 Zygote 与 system_server 之间的密切关系。

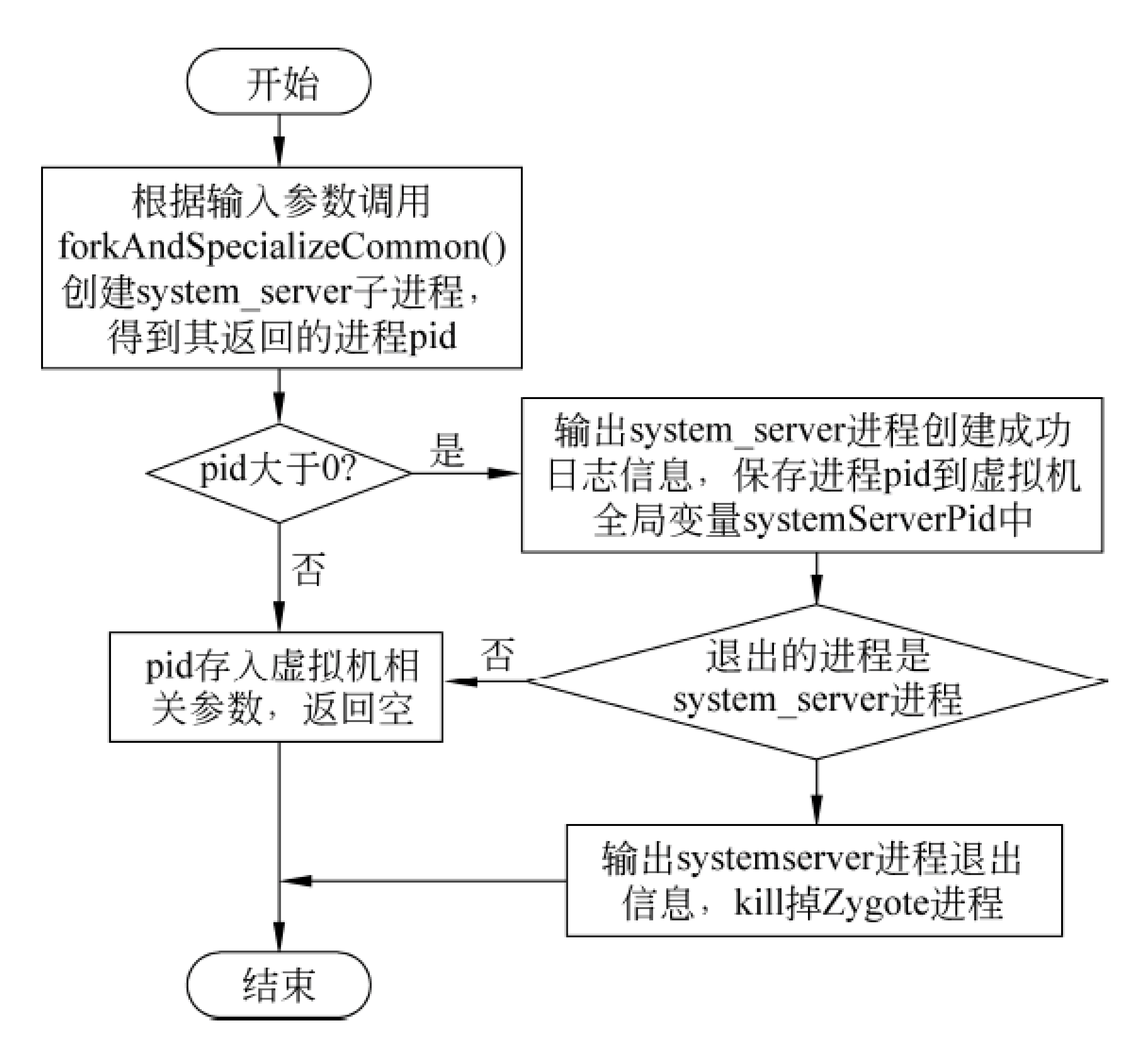


图 6.8 Dalvik_dalvik_system_Zygote_forkSystemServer()函数流程图

(3) forkAndSpecialize()，创建一个非 Zygote 进程，其实现对应的函数为 Dalvik_dalvik_system_Zygote_forkAndSpecialize()，代码如下。

代码清单 6.6 dalvik/vm/native/dalvik_system_Zygote.cpp：Dalvik_dalvik_system_Zygote_forkAndSpecialize()源代码

```
static void Dalvik_dalvik_system_Zygote_forkAndSpecialize(const u4* args,
    JValue* pResult)
{
    pid_t pid;
    pid= forkAndSpecializeCommon(args, false);
    RETURN_INT(pid);
}
```

forkAndSpecialize()不同于 fork()，它产生的子进程不是 Zygote 进程，也就是说它的子进程不会产生新进程。

Dalvik_dalvik_system_Zygote_forkAndSpecialize()函数流程如图 6.9 所示。

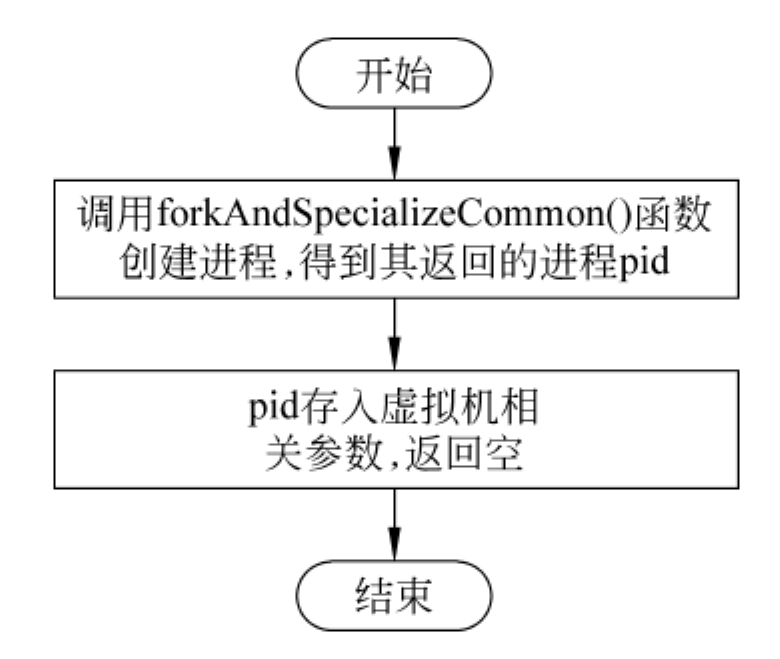


图 6.9 Dalvik_dalvik_system_Zygote_forkAndSpecialize()函数流程图

函数调用 forkAndSpecializeCommon()函数来产生新进程，而在 forkSystemServer()函数中也调用了此函数来产生 system_server 进程，不同的是传递的参数不相同，forkAndSpecializeCommon()函数的第二个参数即标识创建的子进程是否为 system_server 子进程，forkSystemServer()调用时第二个参数为 true，而 forkAndSpecialize()调用时则为 false。

在 Dalvik_dalvik_system_Zygote_forkSystemServer()函数和 Dalvik_dalvik_system_Zygote_forkAndSpecialize()函数中均调用了 forkAndSpecializeCommon()函数来实现产生新进程返回进程 id，其代码如下。

代码清单 6.7　dalvik/vm/native/dalvik_system_Zygote.cpp：forkAndSpecializeCommon () 源代码

```
static pid_t forkAndSpecializeCommon(const u4* args, bool isSystemServer)
{
    pid_t pid;
    uid_t uid= (uid_t) args[0];
    gid_t gid= (gid_t) args[1];
    ArrayObject* gids= (ArrayObject *)args[2];
    u4 debugFlags= args[3];
    ArrayObject * rlimits= (ArrayObject *)args[4];
    int64_t permittedCapabilities, effectiveCapabilities;
        /**判断是否为 system_server 进程 */
    if (isSystemServer) {
        permittedCapabilities= args[5] | (int64_t) args[6] << 32;
        effectiveCapabilities= args[7] | (int64_t) args[8] << 32;
    } else {
        permittedCapabilities= effectiveCapabilities= 0;
    }
        /**判断当前虚拟机是否支持 Zygote */
    if (!gDvm.zygote) {
        dvmThrowIllegalStateException(
            "VM instance not started with -Xzygote");
        return -1;
    }
        /**判断堆创建是否成功,不成功则停止虚拟机 */
    if (!dvmGcPreZygoteFork()) {
        LOGE("pre-fork heap failed");
        dvmAbort();
    }
    /**设置信号处理 */
    setSignalHandler();
    dvmDumpLoaderStats("zygote");
    /** fork 子进程 */
    pid= fork();
```

```
    if (pid==0) {
        /**根据传入的参数对子进程进行一些处理 * /
        int err;
        /**子进程 * /
#ifdef HAVE_ANDROID_OS
        extern int gMallocLeakZygoteChild;
        gMallocLeakZygoteChild=1;
          /**保证 caps 通过 UID 变化,除非处于 root * /
        if (uid != 0) {
            err=prctl(PR_SET_KEEPCAPS, 1, 0, 0, 0);
            if (err < 0) {
                LOGE("cannot PR_SET_KEEPCAPS: %s", strerror(errno));
                dvmAbort();
            }
        }
#endif /**HAVE_ANDROID_OS * /
        /**将 list 数组中所标明的组加入到目前进程的组设置中 * /
        err= setgroupsIntarray(gids);
        if (err < 0) {
            LOGE("cannot setgroups(): %s", strerror(errno));
            dvmAbort();
        }/**设置资源限制 * /
        err= setrlimitsFromArray(rlimits);
        if (err < 0) {
            LOGE("cannot setrlimit(): %s", strerror(errno));
            dvmAbort();
        }
        /**设置指定进程组标志号 * /
        err= setgid(gid);
        if (err < 0) {
            LOGE("cannot setgid(%d): %s", gid, strerror(errno));
            dvmAbort();
        }
        /**设置用户标志号 * /
        err= setuid(uid);
        if (err < 0) {
            LOGE("cannot setuid(%d): %s", uid, strerror(errno));
            dvmAbort();
        }
        int current=personality(0xffffFFFF);
        int success=personality((ADDR_NO_RANDOMIZE | current));
        if (success==-1) {
    LOGW("Personality switch failed. current=%d error=%d\n", current, errno);
        }
```

```
        /**设置 Linux 功能标识 * /
        err= setCapabilities(permittedCapabilities, effectiveCapabilities);
        if (err != 0) {
            LOGE("cannot set capabilities (%llx,%llx): %s",
            permittedCapabilities, effectiveCapabilities, strerror(err));
            dvmAbort();
        }
         /**我们的系统线程 ID已经改变,重新获取新的 * /
        Thread* thread= dvmThreadSelf();
        thread- > systemTid= dvmGetSysThreadId();
        /**配置其余的调试选项 * /
        enableDebugFeatures(debugFlags);
           /**将信号量机制设为默认 * /
        unsetSignalHandler();
           /**子进程不支持 Zygote* /
        gDvm.zygote= false;
           /**检查虚拟机初始化是否成功 * /
        if (!dvmInitAfterZygote()) {
            LOGE("error in post- zygote initialization");
            dvmAbort();
        }
    } else if (pid> 0) {
        /* the parent process * /
        /**父进程 * /
    }
    return pid;
}
```

函数执行流程如图 6.10 所示。

forkAndSecializeCommon()函数与 Dalvik_dalvik_system_Zygote_fork()函数的流程相似：在调用 fork()之前，检查当前进程是否支持 Zygote 功能并进行空堆栈的创建，调用 unsetSignalHandler()函数设置信号处理机制，通过判断 fork()返回的 pid 值来确定是子进程还是父进程。不同之处在于：①在 forkAndSpecializeCommon()函数开始时，需要分析传递过来的参数 isSystemServer，判断所需创建的子进程是否为 system_server，对其参数进行相应处理。②由于调用函数 forkAndSpecializeCommon()所产生的子进程都不是 Zygote 进程，因而，在子进程中需要调用 unsetSignalHandler()函数重置信号机制，同时，子进程是不能再次创建新进程的，故把子进程标志为不支持 Zygote：gDvm. zygote＝false。③在子进程中还要检查虚拟机初始化是否成功，由于所创建的子进程需要执行相应任务，因此必须保证虚拟机成功初始化。

点拨　在 Dalvik 虚拟机中，很多方法均通过 JNI 机制进行调用，在 Java 层定义函数，具体实现采用 C/C++ 编写，采用 C/C++ 实现可以调用 C/C++ 特有的函数，更好地与硬件、操作系统进行交互，提高程序的性能。有关 JNI 机制见第 2 卷第 3 章。

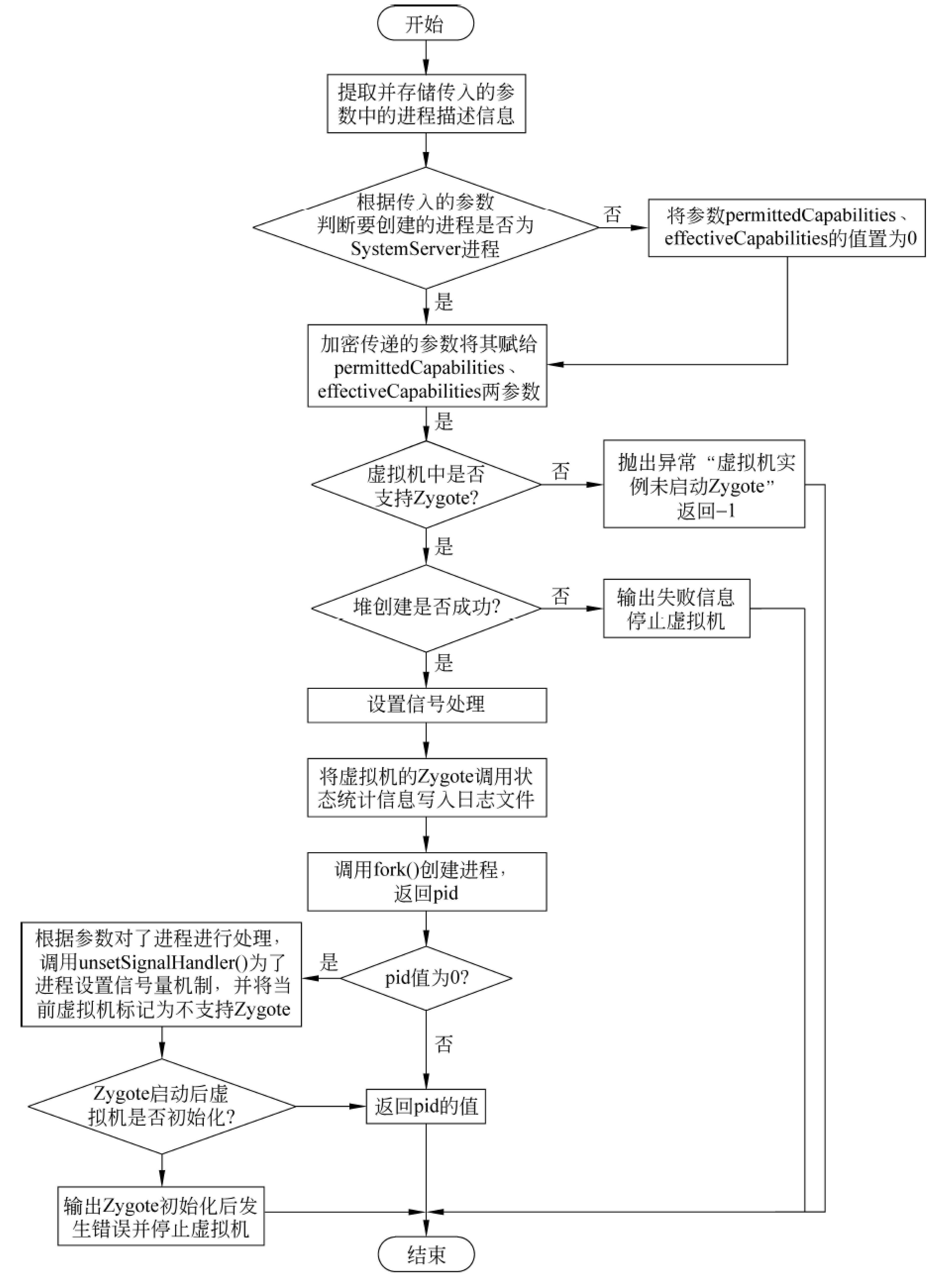


图 6.10 forkAndSecializeCommon()函数流程图

6.4　Dalvik 虚拟机运行应用程序过程

6.4.1　apk 文件生成

在 Android 系统中运行的是专有的 Dex 文件格式，第 3 章已经介绍了 Dex 文件的格式，因此，在这里简单介绍应用程序的生成 Dex 文件并且打包为 apk 文件的过程。

Android 应用开发和 Dalvik 虚拟机 Android 应用所使用的编程语言是 Java 语言，和 Java SE 一样，编译时使用 Sun JDK 将 Java 源程序编译成标准的 Java 字节码文件(.class 文件)，而后通过工具软件 dx 把所有的字节码文件转成 Dex 文件(classes.dex)。最后使用 Android 打包工具(aapt)将 Dex 文件、资源(Resource)文件以及 AndroidManifest.xml 文件(二进制格式)组合成一个应用程序包(apk)。应用程序包可以被发布到手机上运行，如图 6.11 所示。

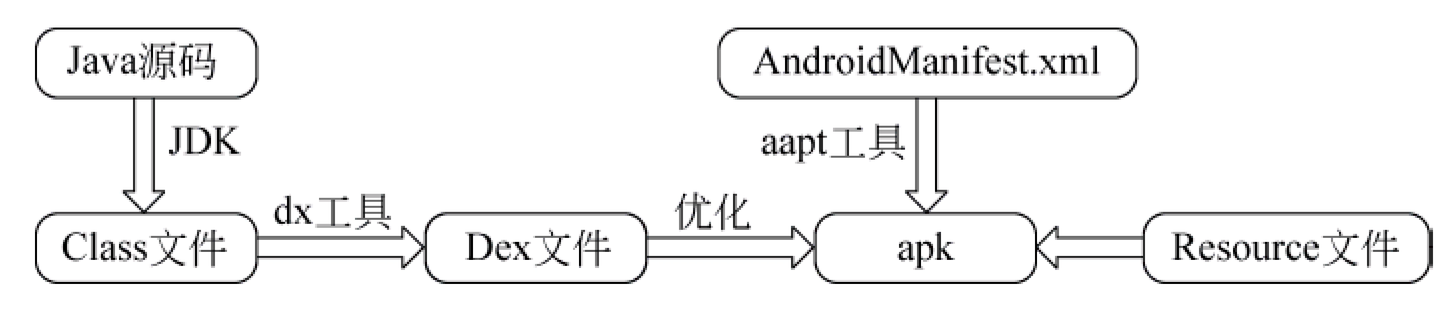


图 6.11　应用程序打包过程图

其中涉及的工具如下。

dx 工具：将 Java 编译后的 class 文件转换成 Dex 格式文件。

Dex 优化：Dex 文件的结构是紧凑的，如果要求运行时的性能有进一步提高，仍然需要对 Dex 文件进行进一步优化，优化后以 odex 结尾。

6.4.2　Dalvik 虚拟机运行应用程序的主要流程

Dalvik 虚拟机运行过程包括初始化运行环境、初始化虚拟机线程、加载核心类和 main 方法类以及执行方法字节码等几个阶段，其运行过程如图 6.12 所示(注：图中流程简要标注了所涉及模块部分，不能作为严格划分标准)。

图 6.12 中标出了流程中的几个阶段所涉及的 Dalvik 虚拟机的相应功能模块。

应用程序是以 apk 文件的形式被虚拟机通过类加载模块引用加载并提取可执行代码的。apk 文件解压后得到 Dex 文件，类加载模块对 Dex 文件进行解析，将所需的类的各个信息抓取出来并将其封装到一个数据结构实例对象中，以供解释器直接引用这个结构体对象的相关的成员变量，实现程序的实际运行。

类加载模块的工作主要分为以下两个阶段。

(1) 取得 Dex 原始文件并将其还原到内存中并将 Dex 文件与一个 DexFile 结构体对象关联。

(2) 对 Dex 文件中的各个类依次进行加载生成类对象实例，并将对象指针交付给解释器引用执行。

在 Dalvik 上执行的程序由字节码组成，由解释器负责解释执行 Dex 字节码。在字节码加载已经完毕后，Dalvik 虚拟机解释器被调用开始取指解释字节码。并根据指令进行相应

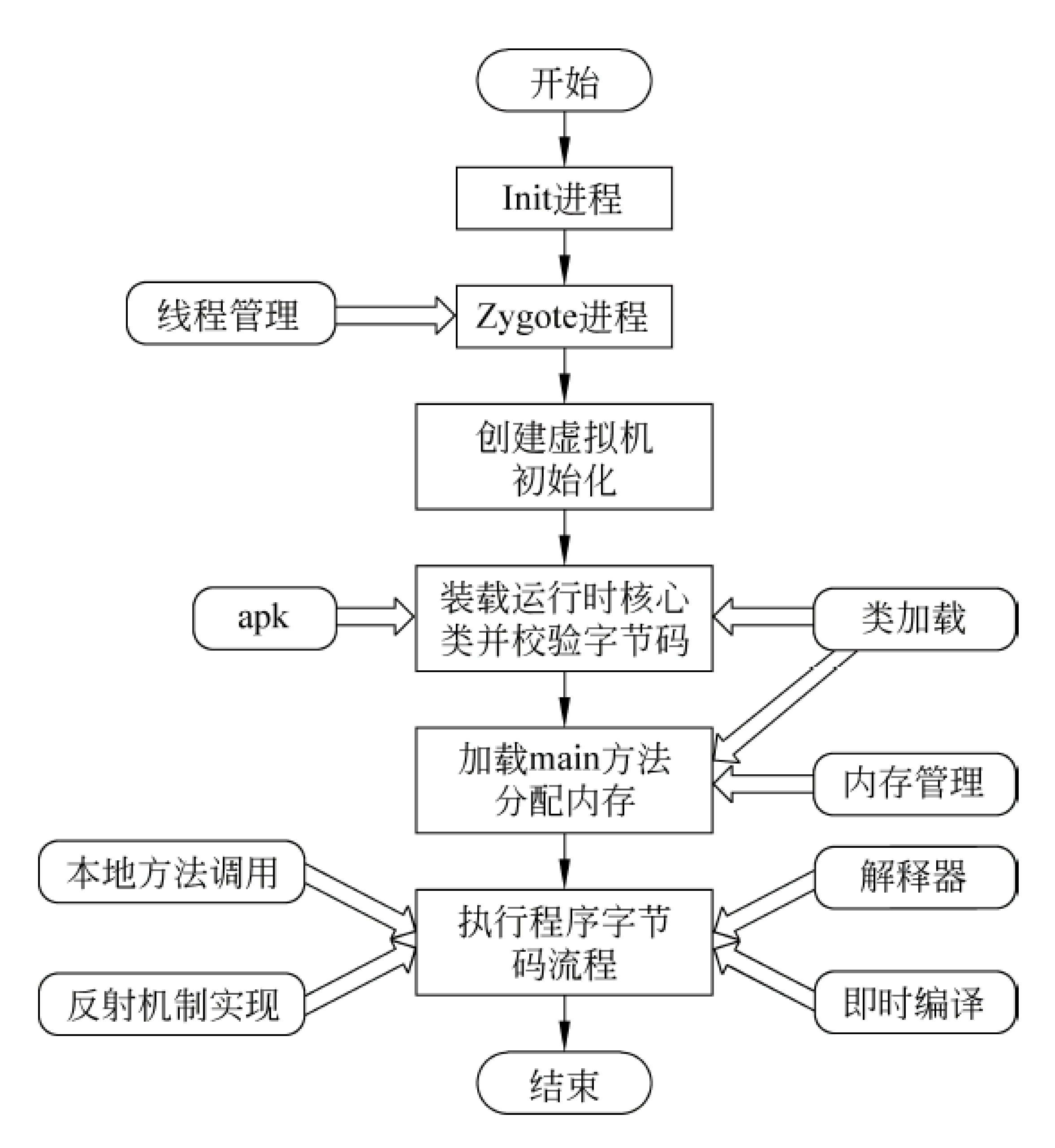


图 6.12 Dalvik 虚拟机运行过程图

的操作，然后返回相应的结果。在 Android 4.04 版本中，解释器共有两种实现，分别是 C 语言实现和汇编语言的实现，也分别称作移动型(Portable)解释器和快速型(Fast)解释器。

为了缓解由于解释器解释带来的低效，当前通常采用两种方法解决这个问题：①采用 NDK，调用静态编译的方法提高速率；②使用 JIT，在运行时编译字节码并优化。虽然这两种方法都能得到不错的效果，但是 JIT 更具一般性和可移植性。JIT 混合了两种技术，解释器解释时，编译部分程序，并在下次直接执行该编译后的源程序。JIT 和解释器之间有非常紧密的联系。

本地方法调用(Java Native Interface，JNI)作为 Java 代码和本地代码互相调用以及交互的桥梁：①Java 程序中的函数调用 Native 语言写的函数，Native 一般指的是 C/C++ 编写的函数；②Native 程序中的函数调用 Java 层的函数，也就是说在 C/C+程序中可以调用 Java 函数。

反射机制允许程序在运行时透过 Reflection API 取得任何一个已知名称的类的内部信息，包括其描述符、超类、实现的接口，也包括属性和方法等所有信息，并可于运行时改变属性内容或调用内部方法。类反射机制在数据库方面的应用更是非常广泛。同时，反射机制具有很好的派生性，例如，Java 语言基于类反射技术还开发提供了动态代理和元数据两种机制，使程序逻辑更加简单，安全性更高。

通过在 Dalvik 虚拟机实际运行场景下，使用 GDB 调试平台输出的堆栈信息，生成函数调用图，进而可以得到各个模块之间的关系图。

点拨 GDB 是 GNU 的程序调试工具，主要完成以下 4 个功能。

(1) 启动程序，指明可能影响其行为的事件。

(2) 使程序停在断点处。

(3) 当程序停止时，检查所发生的事情。

(4) 改变程序的执行环境，以便纠正一个错误，继而了解其他信息。

使用 GDB 调试程序，借助 GDBSERVER，以达到调试 Android 虚拟机或手机中程序的目的。具体做法为：应用程序运行在目标平台，在目标平台运行 GDBSERVER，本地使用 GDB 来调试程序。

小结

Dalvik 虚拟机是 Android 系统的重要组成，Android 系统的执行基础，为应用程序提供运行环境。在 Dalvik 虚拟机中通过解释器进行解析执行应用程序。应用程序虽是使用 Java 语言编写，但其在进入 Dalvik 虚拟机执行前，需要被转换为 Dalvik 专有执行格式——Dex 文件，通过类加载加载到虚拟机，在其中以 Dalvik 字节码的形式被解释器取指执行，应用程序依赖于各个模块的协调工作。通过 Dalvik 虚拟机的执行流程的简要分析对整体的执行流程以及各个模块的位置拥有更直观的认识，有助于对于各个模块的分析的深入理解。